Interval Linear Constraint Solving Using the Preconditioned Interval Gauss-Seidel Method

C.K. Chiu and J.H.M. Lee
Department of Computer Science
The Chinese University of Hong Kong
Shatin, Hong Kong
{ckchiu,jlee}@cs.cuhk.hk

Abstract

We propose the use of the preconditioned interval Gauss-Seidel method as the backbone of an efficient linear equality solver in a CLP(Interval) language. The method, as originally designed, works only on linear systems with square coefficient matrices. Even imposing such a restriction, a naive incorporation of the traditional preconditioning algorithm in a CLP language incurs a high worst-case time complexity of $O(n^4)$, where n is the number of variables in the linear system. In this paper, we generalize the algorithm for general linear systems with m constraints and n variables, and give a novel incremental adaptation of preconditioning of $O(n^2(n+m))$ complexity. The efficiency of the incremental preconditioned interval Gauss-Seidel method is demonstrated using large-scale linear systems.

1 Introduction and Related Work

Interval narrowing [7] is deficient in handling general systems of constraints over real numbers [5, 2]. Chiu and Lee [6] suggest to separate linear equality constraint solving from inequality and non-linear constraint solving. They propose the implementation of a linear solver based on an interval version of Gaussian elimination using generalized interval arithmetic operators [10]. The solver is incorporated in a new interval constraint logic programming system CIAL, which compares favorably against other CLP(Interval) systems, such as CLP(BNR) [3] and Echidna [18]. Later experiments [5] show, however, that the generalized interval Gaussian elimination solver is deficient in solving large-scale linear systems, both in terms of accuracy and computational efficiency. In this paper, we propose the use of the preconditioned interval Gauss-Seidel method [15] as the backbone of an efficient linear equality solver in a CLP(Interval) language.

The interval Gauss-Seidel method is an iterative method for solving interval linear systems. Convergence of the method to the approximations of the hull of the solution set[1] is guaranteed, however, for only a certain class of linear systems. Preconditioning [9] is a process to transform a linear system into another equivalent system satisfying the convergence criteria.

To incorporate the preconditioned interval Gauss-Seidel method into the constraint solver of a CLP language, we have to tackle two problems. First, the method, as originally designed, works only on linear systems with square coefficient matrices.

[1] In general, the hull of the solution set cannot be exactly computed on computer due to the limitation of floating-point representations.

In the context of constraint logic programming, however, there are not always as many variables as constraints. Thus, the traditional method should be modified to handle general linear systems. Second, a good constraint solver must be amenable to efficient incremental execution. The application of preconditioning involves multiplication between preconditioner and coefficient matrix of a linear system, which is of $O(r^2 n)$ complexity where r and n are the number of collected constraints and the number of variables respectively. Even considering only the restricted linear systems with square coefficient matrices of rank n, a naive adaptation of preconditioning for incremental execution in a CLP language incurs a high worst-case time complexity of $O(n^4)$. In this paper, we present an incremental preconditioning algorithm of $O(n^2(n+m))$ complexity for general linear systems with m constraints and n variables. A heuristic for detecting redundant and inconsistent constraints is also presented.

Interval arithmetic is introduced by Moore [13]. Cleary [7] gives a relational version of interval arithmetic for use in Prolog. He describes distinct algorithms, one for each kind of convex constraint over intervals, that narrow intervals associated with a constraint by removing values that do not satisfy the constraint. A constraint relaxation cycle is needed to coordinate the execution of the narrowing algorithms for a network of constraints. We call this technique *interval narrowing*, which effects the development of such systems as BNR-Prolog [16], CLP(BNR) [3], Echidna [19], and ICL [11]. Chiu and Lee [6] improve interval linear constraint solving by using an interval version of Gaussian elimination with generalized interval arithmetic. Benhamou *et al* [2] replaces the usual interval reduction operator of previous CLP(Interval) languages by an operator based on the interval Newton method to speed up interval non-linear constraint solving.

The paper is organized as follows. Section 2 defines notations and reviews the traditional preconditioned interval Gauss-Seidel method. Section 3 presents our incremental version and its correctness and complexity results. In particular, the incremental application of preconditioning is discussed in details. Section 4 describes the benchmarking results. Section 5 summarizes our contributions and sheds light on further work.

2 Notations and Preliminaries

To facilitate subsequent discussions, we fix some notations. We denote mathematical interval variables (or constants) referring to non-empty floating-point intervals by upper (or lower) case letters with superscript I. Real variables (or constants) are denoted by ordinary upper (or lower) case letters. Upper case letters in boldface denote matrices, e.g. $\mathbf{A} = (a_{ij}), \mathbf{B^I} = (b^I_{ij})$, etc. Column vectors are denoted by arrowed letters, such as $\vec{X} = (X_1, \ldots, X_n)^T$, where the superscript T indicates the transpose of a matrix.

If J^I is a non-empty real interval, the *outward-rounding function* $\xi : I(\mathbb{R}) \rightarrow I(\mathbb{F})$ is defined as

$$\xi(J^I) = \bigcap \{ J^{I'} \in I(\mathbb{F}) \mid J^I \subseteq J^{I'} \}$$

where $I(\mathbb{R})$ and $I(\mathbb{F})$ are the sets of real intervals and floating-point intervals respectively. The outward-rounding function gives the tightest floating-point interval containing J^I.

If $\cdot \in \{+, -, \times, /\}$, the corresponding *floating-point interval operations* are denoted by

$$A^I \odot B^I = \{a \cdot b \mid a \in A^I, b \in B^I\}.$$

In the case of interval division, $\oslash$, we assume B^I does not contain 0. Unless otherwise stated, results of interval operations are outward-rounded.

2.1 The Basic Interval Gauss-Seidel Method

In many applications, we have some crude bounds on the solution of a linear system $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$. Such a system can be solved efficiently by using some iterative methods. Interval Gauss-Seidel method is one being widely-used in interval computation [15].

Let the i-th equation in $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$ be

$$\sum_{j=1}^{n} (a^I_{ij} \otimes X^I_j) = b^I_i \,{}^2 \qquad \text{where } i \in \{1, \ldots, n\}$$

and we have initial bounds on all variables. The interval Gauss-Seidel method works by updating each variable X^I_i by

$$X^I_i \leftarrow ((b^I_i \ominus \sum_{j=1, j \neq i}^{n} (a^I_{ij} \otimes X^I_j)) \oslash a^I_{ii}) \bigcap X^I_i$$

in an iterative fashion. If, at any step, any variable becomes the empty interval, then we conclude that the system has no solution.

In an iterative method, a system usually takes more than one iterating cycle to converge. In addition, since we are considering constraint solving in a single processor machine, only one equation can be examined at a time in sequence. The previously computed values can be used as soon as they are available. Assuming that variable updates are coordinated in a naive round-robin fashion, a sequential version of the interval Gauss-Seidel method is suggested to be [1]

$$X^{I(k)}_i \leftarrow ((b^I_i \ominus \sum_{j=1}^{j=i-1} (a^I_{ij} \otimes X^{I(k)}_j) \ominus \sum_{j=i+1}^{j=n} (a^I_{ij} \otimes X^{I(k-1)}_j)) \oslash a^I_{ii}) \bigcap X^{I(k-1)}_i.$$

The superscript $(k-1)$ of $X^{I(k-1)}_i$ indicates that the variable is obtained in the $(k-1)$-st iterating cycle. The interval Gauss-Seidel method terminates when all variables remain unchanged after an iteration or when the difference between the new and last computed value of each variable is less than a user-defined number.

The following definitions and lemma state the convergence criterion of a linear system using the interval Gauss-Seidel method.

Definition 2.1 [15] *The* hull of the solution set *of a linear system is the set of tightest intervals that enclose the solution of the linear system.* ■

Definition 2.2 [15] *The* magnitude *of an interval* $I^I = [l, u]$ *is defined as* $mag(I^I) = max(|l|, |u|)$, *while its* mignitude *is defined as* $mig(I^I) = min(|l|, |u|)$, *where* $|a|$ *denotes the absolute value of a real number* a. *An interval matrix* $\mathbf{A^I} = (a^I_{ij})$ *is said to be* strictly diagonal dominant *if*

$$mig(a^I_{ii}) > \sum_{k=1, k \neq i}^{n} mag(a^I_{ik}) \quad for \quad i = 1, \ldots, n. \qquad ■$$

[2] We overload the $\sum$ symbol to denote interval summation.

Lemma 2.3 [15] *Interval Gauss-Seidel method is guaranteed to converge to the hull of the solution set*[3] *of a linear system if the coefficient matrix of the linear system is strictly diagonal dominant.* ■

2.2 Preconditioning

As stated in lemma 2.3, the interval Gauss-Seidel method always converges to the hull of the solution set on a system with strictly diagonal dominant coefficient matrix, but this criterion may not be satisfied in a general system. Hence, one may attempt to transform the system into an equivalent system in the sense that the new system contains all solutions of the original system, but is strictly diagonal dominant. *Preconditioning* effects such a transformation.

Preconditioning is usually done by multiplying a suitable *point* matrix $\mathbf{P}$ to the original system. Instead of solving $\mathbf{A^I} \otimes \vec{X^I} = \vec{b^I}$, we deal with the following system:

$$\mathbf{P} \otimes \mathbf{A^I} \otimes \vec{X^I} = \mathbf{P} \otimes \vec{b^I}. \tag{1}$$

We call $\mathbf{P}$ the *preconditioner*. Hansen [9] suggests an inverse mid-point matrix as preconditioner, which is shown to be optimal [4] in the sense that the preconditioned system gives the tightest bounds of the solutions of the original system. Let $\mathbf{\check{A}}$ denote the mid-point matrix of $\mathbf{A^I}$. We define

$$\begin{array}{lll} \check{a}_{ij} = (l_{ij} + u_{ij})/2 & \text{where} & \mathbf{A^I} = ([l_{ij}, u_{ij}]) \text{ and } \mathbf{\check{A}} = (\check{a}_{ij}) \\ & \text{for} & i, j = 1, 2, \ldots, n. \end{array}$$

We compute the inverse of $\mathbf{\check{A}}$ using, say row reduction, in high precision floating-point arithmetic. The real $\mathbf{\check{A}^{-1}}$ is used as the preconditioner $\mathbf{P}$ in equation (1).

Since preconditioning involves much interval multiplications, small errors will be introduced due to outward-rounding. A preconditioned system usually has slightly wider solutions than the original system and these additional pseudo-solutions are called overestimation [14]. Overestimation destroys the completeness of inconsistency detection in interval Gauss-Seidel method since an inconsistent system of constraints may become consistent after preconditioning. Readers may refer to [14, 15] for detailed analysis.

3 Incremental Execution

Solvers in constraint logic programming languages must be amenable to efficient incremental execution. The preconditioned interval Gauss-Seidel method, as originally designed, however, operates in the batch mode: all constraints are collected before solving takes place. To adapt the preconditioned interval Gauss-Seidel method for incremental execution, we need to consider the incremental *update of preconditioner*, *detection of inconsistency and redundancy*, and *application of preconditioning*. We present the details of the stated issues in the following.

3.1 Incremental Update of Preconditioner

We adopt the optimal preconditioner, the inverse mid-point matrix, as stated in section 2.2. Assume that we have a collection of m interval linear equalities of n

[3] If floating-point interval arithmetic is employed, the solutions obtained are usually slightly wider than the hull of the solution set since outward-rounding is made.

variables. The mid-point coefficient matrix $\check{\mathbf{A}}$ is thus an $m \times n$ matrix. The entries in $\check{\mathbf{A}}$, which are mid-points of intervals, cannot be represented exactly on computer in general. We can simply round them to their nearest floating-point number. The small errors introduced in the mid-point matrix do not affect the convergence of the preconditioned system significantly.

Constraints are generated and submitted to the constraint solver incrementally in a constraint logic programming system. The linear system present in the solver do not necessarily have a square matrix in general. We present how the preconditioner can be computed from such a rectangular matrix.

For the case where $n < m$, it implies that some equalities are either redundant or inconsistent to the system. Those equalities will be located by another algorithm using heuristic (to be discussed in section 3.2) and they should not be used in the calculation of the inverse. We disregard this case.

Otherwise, we have $n \geq m$. We define a corresponding *rectangular identity matrix* $\mathbf{J}$ by

$$\begin{array}{rll} \mathbf{J} = (j_{kl}), & \text{where} & j_{kl} = 1 \text{ for } k = l, \quad j_{kl} = 0 \text{ for } k \neq l \\ & \text{for} & 1 \leq k \leq m \text{ and } 1 \leq l \leq n. \end{array}$$

The preconditioner $\mathbf{P}$ is computed by row reducing the combined matrix $[\check{\mathbf{A}}|\mathbf{J}]$ until the the first m columns of $\check{\mathbf{A}}$ becomes the identity matrix $\mathbf{I}$. The required preconditioner $\mathbf{P}$ resides in the first m columns of the original $\mathbf{J}$ matrix[4]. Therefore, the row reduced matrix has the form $[\mathbf{I}|\mathbf{U}|\mathbf{P}|\mathbf{Z}]$, where $\mathbf{I}$ is the $m \times m$ identity matrix, $\mathbf{U}$ is an $m \times (n-m)$ matrix to be used for future update of the preconditioner, $\mathbf{Z}$ is an $m \times (n-m)$ zero matrix. We call $[\mathbf{I}|\mathbf{U}|\mathbf{P}|\mathbf{Z}]$ the *IUPZ matrix*.

To construct the IUPZ matrix incrementally, we adapt a variant of the familiar incremental Gaussian elimination procedure used in CLP($\mathcal{R}$) [12]. Assume that we have a collection of r interval linear equalities of n variables with $r < n$. When a new linear equality, whose mid-point coefficients are denoted by $m_{r+1,l}$, is added, the IUPZ matrix, $\mathbf{X} = (x_{kl})$ for $1 \leq k \leq r, 1 \leq l \leq n$, is updated incrementally as shown in algorithm 1. The calculation should be performed in high precision floating-point arithmetic. The updated $(r+1) \times (r+1)$ preconditioner resides in the $(n+1)$-st to $(n+r+1)$-st columns of the IUPZ matrix $\mathbf{X}$.

3.2 Detection of Inconsistency and Redundancy

There is no general method to detect redundancy in an iterative method, especially in the interval context. Inconsistency is revealed if an empty intersection is produced by applying preconditioned Gauss-Seidel method directly to the system. However, overestimation prevents us from detecting all possible inconsistency.

In the implementation of an interval linear constraint solver, we *must not* use inconsistent or redundant constraints in computing the preconditioner. Intuitively, a system with redundant or inconsistent constraints often have more constraints than the number of unknown variables. Selecting the "wrong" subset of equalities for preconditioning produces a poor preconditioner in the sense that the preconditioned system may fail to converge.

We use a simple heuristic to locate inconsistent and redundant equalities. Incremental calculation of a matrix inverse involves forward substitution. If we find that after forward substitution on, say, the $(r+1)$-st constraint during the calculation of the inverse of the mid-point coefficient matrix $\check{\mathbf{A}}$, all its remaining $n-r$ coefficient

[4] Note that $\mathbf{P}$ is an $m \times m$ matrix.

1. The IUPZ matrix $\mathbf{X}$ is augmented to $r+1$ rows by appending an extra row where

$$\begin{array}{rcll} x_{r+1,l} & = & m_{r+1,l} & \text{for} \quad 1 \leq l \leq n \\ x_{r+1,l} & = & 0 & \text{for} \quad n+1 \leq l \leq 2n \text{ and } l \neq n+r+1 \\ x_{r+1,l} & = & 1 & \text{for} \quad l = n+r+1. \end{array}$$

2. (Forward substitution) We subtract the $(r+1)$-st row of $\mathbf{X}$ from suitable multiples of the first r rows such that the first r columns of the $(r+1)$-st row become zeros.
3. (Row Normalization) We divide all the elements at the $(r+1)$-st row by a suitable value such that $x_{r+1,r+1}$ becomes 1.
4. (Backward substitution) We subtract the first r rows of $\mathbf{X}$ from suitable multiples of the $(r+1)$-st row such that the $(r+1)$-st columns of the first r rows become zeros.

Algorithm 1: Incremental Update Procedure for the IUPZ Matrix

mid-points are less than a small user-defined value, say 10^{-10}, then we conclude that the $(r+1)$-st constraint is either "redundant" or "inconsistent." Inconsistent and redundant constraints are both regarded as *fruitless to preconditioning* and will not be employed in the preconditioning process.

Note that our proposed method is only a heuristic. Constraints concluded to be redundant or inconsistent may indeed be independent and consistent. We cannot simply conclude that the system is inconsistent. It is also improper to disregard these constraints since it can result in excessively relaxed answer constraints. Current practice in the CIAL system is to transfer these constraints to another solver, which employs interval narrowing for constraint solving, for further scrutiny, in the hope that the solver may narrow some intervals or reveal inconsistency. Thus, computation results of CIAL are interpreted as conditional answers [20, 5].

Lemma 3.1 *Assume that we have a linear system with m equalities of n variables. The incremental preconditioner update algorithm with inconsistency and redundancy detection has worst-case time complexity $O(n^2(n+m))$.*

Proof: We consider the most complicated case where there are n independent and consistent constraints ("independent constraints" for short), and $(m-n)$ inconsistent/redundant constraints. The incremental algorithm is invoked once a constraint is added.

Assume that there are already i independent constraints in the solver. The addition of a new independent constraint involves $(2n-i+1)ic_1$ operations in forward substitution, $(2n-i)c_2$ operations in row normalization, and $(2n-i)ic_1$ operations in backward substitution, where c_1 and c_2 are constants and denote the cost of each operation in forward substitution and normalization respectively. The incremental update algorithm thus takes $\sum_{i=0}^{n-1}((4n-2i+1)ic_1+(2n-i)c_2)$ operations to add n independent constraints.

$$
\underset{\mathbf{P}_{(r+1)\times(r+1)}}{\left[\begin{array}{c|c} \mathbf{O}_{r\times r} & \mathbf{L}_{r\times 1} \\ \hline \mathbf{M}_{1\times r} & \mathbf{N}_{1\times 1} \end{array}\right]}
\otimes
\underset{\mathbf{A^I}_{(r+1)\times n}}{\left[\begin{array}{c|c} \mathbf{A^I_1}_{r\times r} & \mathbf{A^I_2}_{r\times(n-r)} \\ \hline \mathbf{A^I_3}_{1\times r} & \mathbf{A^I_4}_{1\times(n-r)} \end{array}\right]}
=
\underset{\mathbf{P}_{(r+1)\times(r+1)} \otimes \mathbf{A^I}_{(r+1)\times n}}{\left[\begin{array}{c|c} \mathbf{R^I_1}_{r\times r} & \mathbf{R^I_2}_{r\times(n-r)} \\ \hline \mathbf{R^I_3}_{1\times r} & \mathbf{R^I_4}_{1\times(n-r)} \end{array}\right]}
$$

Figure 1: Partitioning of $\mathbf{P}$, $\mathbf{A^I}$, and $\mathbf{P} \otimes \mathbf{A^I}$

The detection of an inconsistent or redundant constraint takes $(2n - i + 1)ic_1$ operations since it involves only forward substitution. In the *worst* case, all inconsistent/redundant constraints arrive after $(n-1)$ independent constraints have been added[5]. The heuristic thus takes $(2n - (n-1) + 1)(n-1)c_1(m-n)$ operations to locate $(m-n)$ inconsistent/redundant constraints.

The complexity of the incremental preconditioner update algorithm with inconsistency and redundancy detection has worst-case time complexity

$$(n+2)(n-1)(m-n)c_1 + \sum_{i=0}^{n-1}((4n-2i+1)ic_1 + (2n-i)c_2) = O(n^2(n+m)). \quad \blacksquare$$

3.3 Incremental Application of Preconditioning

We apply preconditioning only in the cases where the number of variables is more than or equal to the number of constraints[6]. The application of preconditioning involves multiplication of a point preconditioner matrix and an interval coefficient matrix, which is of $O(r^2n)$ complexity[7], where r and n are the number of collected constraints and the number of variables respectively. Without incremental application, we need to re-compute the preconditioned system from scratch whenever new equalities are added in a derivation step. In the worst case (i.e. when only exactly *one* new equality is collected in each derivation step and the entire preconditioner is modified), the whole preconditioning application has complexity $O(n^4)$. We give an incremental adaptation of preconditioning application of order $O(n^3)$.

Let $\mathbf{O}' \otimes \mathbf{A}'^{\mathbf{I}} \otimes \vec{X}^I = \mathbf{O}' \otimes \vec{b}'^I$ be a preconditioned system with r constraints and n variables, where $r < n$. Without loss of generality, we assume that one new equality is added to the system. *Our goal is to make use of the previously calculated* $\mathbf{O}' \otimes \mathbf{A}'^{\mathbf{I}}$ *and* $\mathbf{O}' \otimes \vec{b}'^I$ *to compute some parts of the new preconditioned system* $\mathbf{P} \otimes \mathbf{A^I} \otimes \vec{X}^I = \mathbf{P} \otimes \vec{b}^I$. Note that $\mathbf{A^I}$ differs from $\mathbf{A}'^{\mathbf{I}}$ by only an extra last row.

Consider the computation of $\mathbf{P} \otimes \mathbf{A^I}$. We partition the new preconditioner $\mathbf{P}$, the new coefficient matrix $\mathbf{A^I}$, and their product as shown in figure 1. The product

[5] It maximizes the value of the function $f(i) = (2n - i + 1)ic_1$ where $i \in \{1, \ldots, n-1\}$, which indicates the number of operations.

[6] Extra constraints should have been moved to another solver in the phase of inconsistency and redundancy detection.

[7] We do not consider such special divide-and-conquer *square* matrix multiplication algorithms as Strassen's algorithm ($O(n^{2.81})$) [8]. Those algorithms usually introduce multiple occurrences of variables and require the dimension of the matrix to be a power of 2. The latter can double the storage in the worst case.

matrix $\mathbf{R^I}$ can be calculated by the following equations:

$$\begin{array}{rclcrcl} \mathbf{R^I_1} &=& \mathbf{O} \otimes \mathbf{A^I_1} \oplus \mathbf{L} \otimes \mathbf{A^I_3} & & \mathbf{R^I_2} &=& \mathbf{O} \otimes \mathbf{A^I_2} \oplus \mathbf{L} \otimes \mathbf{A^I_4} \\ \mathbf{R^I_3} &=& \mathbf{M} \otimes \mathbf{A^I_1} \oplus \mathbf{N} \otimes \mathbf{A^I_3} & & \mathbf{R^I_4} &=& \mathbf{M} \otimes \mathbf{A^I_2} \oplus \mathbf{N} \otimes \mathbf{A^I_4} \end{array}. \quad (2)$$

All terms in (2), except $\mathbf{O}\otimes\mathbf{A^I_1}$ and $\mathbf{O}\otimes\mathbf{A^I_2}$, can be calculated in $O(r^2)$ or $O(r(n-r))$ time. We concentrate on the computation of $\mathbf{O} \otimes \mathbf{A^I_1}$ and $\mathbf{O} \otimes \mathbf{A^I_2}$.

The row reduced IUPZ matrix has the form $[\mathbf{I}|\mathbf{U}|\mathbf{O}'|\mathbf{Z}]$ before a new constraint is added. Let $\mathbf{O}^{'} = (o^{'}_{kl})$ for $1 \leq k, l \leq r$ and (u_k) be the first column of $\mathbf{U}$. When a new constraint is added, the IUPZ matrix is depicted in the following.

$$\left[\begin{array}{cccccc|cccccc} 1 & 0 & \cdots & 0 & u_1 & \cdots & o^{'}_{11} & o^{'}_{12} & \cdots & o^{'}_{1r} & 0 & \cdots \\ 0 & 1 & \cdots & 0 & u_2 & \cdots & o^{'}_{21} & o^{'}_{22} & \cdots & o^{'}_{2r} & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 & u_r & \cdots & o^{'}_{r1} & o^{'}_{r2} & \cdots & o^{'}_{rr} & 0 & \cdots \\ \breve{a}_{r+1,1} & \breve{a}_{r+1,2} & \cdots & \breve{a}_{r+1,r} & \breve{a}_{r+1,r+1} & \cdots & 0 & 0 & \cdots & 0 & 1 & \cdots \end{array}\right]$$

If the new constraint is independent and consistent to the system, we should have the following intermediate state after row normalization of the incremental preconditioner update procedure:

$$\left[\begin{array}{cccccc|cccccc} 1 & 0 & \cdots & 0 & u_1 & \cdots & o^{'}_{11} & o^{'}_{12} & \cdots & o^{'}_{1r} & 0 & \cdots \\ 0 & 1 & \cdots & 0 & u_2 & \cdots & o^{'}_{21} & o^{'}_{22} & \cdots & o^{'}_{2r} & 0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 & u_r & \cdots & o^{'}_{r1} & o^{'}_{r2} & \cdots & o^{'}_{rr} & 0 & \cdots \\ 0 & 0 & \cdots & 0 & 1 & \cdots & t_1 & t_2 & \cdots & t_r & t_{r+1} & \cdots \end{array}\right]$$

where t_i's are some intermediate values. The updated $(r+1) \times (r+1)$ preconditioner $\mathbf{P}$ is

$$\mathbf{P} = \left[\begin{array}{ccccc} o^{'}_{11} - u_1 t_1 & o^{'}_{12} - u_1 t_2 & \cdots & o^{'}_{1r} - u_1 t_r & -u_1 t_{r+1} \\ o^{'}_{21} - u_2 t_1 & o^{'}_{22} - u_2 t_2 & \cdots & o^{'}_{2r} - u_2 t_r & -u_2 t_{r+1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ o^{'}_{r1} - u_r t_1 & o^{'}_{r2} - u_r t_2 & \cdots & o^{'}_{rr} - u_r t_r & -u_r t_{r+1} \\ t_1 & t_2 & \cdots & t_r & t_{r+1} \end{array}\right]. \quad (3)$$

What we have described is the analytic solution of $\mathbf{P}$, which depends by no means on the mode, real or floating-point, of the arithmetic operators. *Assuming that we are using real (interval) arithmetic*, we can establish the following equality and inclusion relationships. We decompose the upper-left $r \times r$ sub-matrix $\mathbf{O}$ of $\mathbf{P}$ as follows.

$$\mathbf{O} = \mathbf{O}^{'} - \begin{pmatrix} u_1 \\ u_2 \\ \cdots \\ u_r \end{pmatrix} \times (t_1\ t_2\ \ldots\ t_r). \quad (4)$$

It follows that $\mathbf{O} \otimes \mathbf{A^I_1}$ and $\mathbf{O} \otimes \mathbf{A^I_2}$ can be approximated[8] by

$$\mathbf{O} \otimes \mathbf{A^I_1} \quad \subseteq \mathbf{O}^{'} \otimes \mathbf{A^I_1} \ominus \begin{pmatrix} u_1 \\ u_2 \\ \cdots \\ u_r \end{pmatrix} \otimes ((t_1\ t_2\ \ldots\ t_r) \otimes \mathbf{A^I_1}) \quad (5)$$

[8] A floating-point number a can be regarded as a point interval $[a, a]$. Thus $\mathbf{O}$ can be regarded as a matrix of point intervals.

$$\mathbf{O} \otimes \mathbf{A_2^I} \quad \subseteq \mathbf{O}^{'} \otimes \mathbf{A_2^I} \ominus \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ u_r \end{pmatrix} \otimes ((t_1 \ t_2 \ \dots \ t_r) \otimes \mathbf{A_2^I}) \tag{6}$$

using the *subdistributivity*[9] and *associativity*[10] [13] properties of interval arithmetic. *Unfortunately, none of (4), (5), and (6) hold under floating-point (interval) arithmetic* since associativity and subdistributivity are no longer guaranteed.

The right-hand-sides of (5) and (6) contain $\mathbf{O}^{'} \otimes \mathbf{A_1^I}$ and $\mathbf{O}^{'} \otimes \mathbf{A_2^I}$, slight supersets of which are available from the previous preconditioned system. The multiplication of an $1 \times r$ vector and an $r \times r$ (or $r \times (n-r)$) matrix is an $O(r^2)$ (or $O(r(n-r))$) operation. Thus the computations of the right-hand-sides of (5) and (6) are of $O(r^2)$ and $O(r(n-r))$ complexity respectively. We adapt this more efficient method to precondition the system instead of using $\mathbf{P}$ as defined in (3). In the following we state the preconditioning procedure *before justifying the correctness of the procedure.*

We first widen each component of the floating-point vector $(u_1, u_2, \dots, u_r)^T$ by a small amount, say $1e^{-12}$. The result is an interval vector $(u_1^I, u_2^I, \dots, u_r^I)^T$. We define $\mathbf{C_1^I}$ and $\mathbf{C_2^I}$ as follows:

$$\mathbf{C_1^I} = \mathbf{O}^{'} \otimes \mathbf{A_1^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes ((t_1 \ t_2 \ \dots \ t_r) \otimes \mathbf{A_1^I})$$

$$\mathbf{C_2^I} = \mathbf{O}^{'} \otimes \mathbf{A_2^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes ((t_1 \ t_2 \ \dots \ t_r) \otimes \mathbf{A_2^I}).$$

We modify the left-hand-side of the preconditioned system by replacing the calculation of $\mathbf{R_1^I}$ and $\mathbf{R_2^I}$ in (2). The new calculation is:

$$\mathbf{R_1^I} \quad = \quad \mathbf{C_1^I} \oplus \mathbf{L} \otimes \mathbf{A_3^I} \qquad\qquad \mathbf{R_2^I} \quad = \quad \mathbf{C_2^I} \oplus \mathbf{L} \otimes \mathbf{A_4^I}.$$

We call the new left-hand-side of the preconditioned system $\mathbf{K^I} \otimes \vec{X^I}$. The next step is to find an appropriate floating-point preconditioner $\mathbf{P}'$ to multiply $\vec{b^I}$, the criterion being that $\mathbf{P}' \otimes_r \mathbf{A^I} \subseteq \mathbf{K^I}$, where the symbol $\otimes_r$ denotes the *real* interval multiplication. We propose $\mathbf{P}'$ to be $\mathbf{P}$ with the $\mathbf{O}$ part (as shown in figure 1) replaced by $\mathbf{O_{new}}$ defined as follows:

$$\mathbf{O_{new}} \in \mathbf{O}^{'} \ominus_i \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes_i (t_1 \ t_2 \ \dots \ t_r) \tag{7}$$

where the symbol $\ominus_i$ and $\otimes_i$ denote the inward-rounded[11] interval subtraction and multiplication respectively.

---

[9] $A^I \otimes (B^I \oplus C^I) \subseteq A^I \otimes B^I \oplus A^I \otimes C^I$.

[10] $(A^I \otimes B^I) \otimes C^I = A^I \otimes (B^I \otimes C^I)$.

[11] If J^I is a non-empty real interval, the *inward-rounding function* $\eta : I(\mathbb{R}) \to I(\mathbb{F})$ is defined as

$$\eta(J^I) = \bigcup \{J^{'I} \in I(\mathbb{F}) | J^{'I} \subseteq J^I\}.$$

Lemma 3.2

$$\begin{aligned}
\mathbf{O_{new}} \otimes_r \mathbf{A_1^I} &\subseteq \mathbf{O}^{'} \otimes \mathbf{A_1^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes ((t_1\ t_2\ \dots\ t_r) \otimes \mathbf{A_1^I}). \\
\mathbf{O_{new}} \otimes_r \mathbf{A_2^I} &\subseteq \mathbf{O}^{'} \otimes \mathbf{A_2^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes ((t_1\ t_2\ \dots\ t_r) \otimes \mathbf{A_2^I}).
\end{aligned}$$

Proof: From equation (7), we have

$$\mathbf{O_{new}} \in \mathbf{O}^{'} \ominus_i \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes_i (t_1\ t_2\ \dots\ t_r).$$

Let the symbols $\otimes_r$ and $\ominus_r$ denote the *real* interval multiplication and subtraction respectively. It follows that

$$\begin{aligned}
\mathbf{O_{new}} \otimes_r \mathbf{A_1^I} &\subseteq (\mathbf{O}^{'} \ominus_i \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes_i (t_1\ t_2\ \dots\ t_r)) \otimes_r \mathbf{A_1^I} \\
&\subseteq (\mathbf{O}^{'} \ominus_r \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes_r (t_1\ t_2\ \dots\ t_r)) \otimes_r \mathbf{A_1^I} \\
&\subseteq \mathbf{O}^{'} \otimes_r \mathbf{A_1^I} \ominus_r (\begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes_r (t_1\ t_2\ \dots\ t_r)) \otimes_r \mathbf{A_1^I} \\
&= \mathbf{O}^{'} \otimes_r \mathbf{A_1^I} \ominus_r \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes_r ((t_1\ t_2\ \dots\ t_r) \otimes_r \mathbf{A_1^I}) \\
&\subseteq \mathbf{O}^{'} \otimes \mathbf{A_1^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes ((t_1\ t_2\ \dots\ t_r) \otimes \mathbf{A_1^I}).
\end{aligned}$$

Similarly, we can show

$$\mathbf{O_{new}} \otimes_r \mathbf{A_2^I} \subseteq \mathbf{O}^{'} \otimes \mathbf{A_2^I} \ominus \begin{pmatrix} u_1^I \\ u_2^I \\ \dots \\ u_r^I \end{pmatrix} \otimes ((t_1\ t_2\ \dots\ t_r) \otimes \mathbf{A_2^I}). \qquad \blacksquare$$

By lemma 3.2, $\mathbf{P}'$ satisfies the criterion that $\mathbf{P}' \otimes_r \mathbf{A^I} \subseteq \mathbf{K^I}$. The definition of $\mathbf{O_{new}}$ also explains why we need to widen the components of the $(u_1, \dots, u_r)^T$

vector: this is to facilitate the computation using inward-rounding so that there will be less chance of "rounding inwardly" into empty intervals. Experiments show that each element in the resultant matrix at the right-hand-side of (7) usually contains several floating-point numbers so that the matrix $\mathbf{O_{new}}$ can be easily found. In the case where some elements in the resultant matrix are empty intervals, we can further widen the vector $(u_1, u_2, \ldots, u_r)^T$.

Therefore, the preconditioned system is $\mathbf{K^I} \otimes \vec{X^I} = \mathbf{P'} \otimes \vec{b^I}$. The following lemma and theorem state the correctness result of our incremental preconditioning procedure.

Lemma 3.3 *Given two systems $\mathbf{A^I} \otimes \vec{X}^I = \vec{b}^I$ and $\mathbf{K^I} \otimes \vec{X}^I = \mathbf{P} \otimes \vec{b}^I$. Assume that $\vec{x_1}^I$ and $\vec{x_2}^I$ are the solutions of the two systems accordingly. If $\mathbf{P} \otimes_r \mathbf{A^I} \subseteq \mathbf{K^I}$, then $\vec{x_1}^I \subseteq \vec{x_2}^I$.*

Proof: Let $\vec{x'}^I$ be the solutions of $\mathbf{P} \otimes_r \mathbf{A^I} \otimes \vec{X}^I = \mathbf{P} \otimes_r \vec{b}^I$. Traditional non-incremental preconditioning guarantees that $\vec{x_1}^I \subseteq \vec{x'}^I$. Since $\mathbf{P} \otimes_r \mathbf{A^I} \subseteq \mathbf{K^I}$ and $\mathbf{P} \otimes_r \vec{b}^I \subseteq \mathbf{P} \otimes \vec{b}^I$, from the inclusion monotonicity of interval arithmetic [13], we know that $\vec{x'}^I \subseteq \vec{x_2}^I$. It follows that $\vec{x_1}^I \subseteq \vec{x_2}^I$. ∎

Theorem 3.4 *Assume that we have a linear system with m equalities of n variables. The incremental preconditioning algorithm has worst-case time complexity of $O(n^2(n+m))$. All solutions of an interval linear system are preserved in the incremental preconditioned system.*

Proof: The algorithm consists of incremental update of preconditioner (with inconsistency and redundancy detection) and incremental application of preconditioning. As stated in lemma 3.1, the preconditioner update procedure has worst-case time complexity of $O(n^2(n+m))$.

The preconditioned system is updated incrementally in either $O(r^2)$ or $O(r(n-r))$ time whenever the $(r+1)$-st new independent equality is added. In the worst case, there are n independent equalities and only one of them is collected in each derivation step. The incremental application of preconditioning procedure thus has time complexity of $\sum_{r=0}^{n-1} r^2$ (or $\sum_{r=0}^{n-1} r(n-r)) = O(n^3)$.

It follows that the whole incremental preconditioning algorithm has worst-case time complexity $O(n^2(n+m))$.

From lemmas 3.2 and 3.3, the incremental preconditioning algorithm preserves all solutions of an interval linear system. ∎

3.4 Interval Propagation

The incremental preconditioning algorithm performs only constraint transformations. The domain of interval variables are then narrowed by the interval Gauss-Seidel method.

In the interval Gauss-Seidel method, interval propagation proceeds unidirectionally from non-diagonal variables to diagonal variables. Such a propagation fails to narrow some variables of systems which have more unknown variables than constraints. To maximize infeasible value elimination, we modify the interval propagation in the interval Gauss-Seidel method as follows. Consider the i-th constraint

$$\sum_{j=1}^{n} (a_{ij}^I \otimes X_j^I) = b_i^I \qquad \text{where } i \in \{1, \ldots, m\}$$

in a linear system of m constraints and n variables, variables are updated by

$$X_k^I \leftarrow ((b_i^I \ominus \sum_{j=1, j \neq k}^{n} (a_{ij}^I \otimes X_j^I)) \oslash a_{ik}^I) \bigcap X_k^I \qquad \text{for } k = i \text{ and } m+1, \ldots, n.$$

The invocation of constraints in the linear constraint solver may not be organized in a round-robin fashion. Readers may refer to [6] for details.

4 Benchmarks

The proposed preconditioned interval Gauss-Seidel method has been incorporated into the CIAL system [6], replacing the previous solver which is based on interval Gaussian-elimination. The CIAL 1.0 (Beta) solver lacks incremental execution. Although the preconditioner is constructed incrementally, the preconditioned system (multiplying the preconditioner and the interval coefficient matrix) are re-computed at every derivation step. The solver in CIAL 1.1 contains the proposed incremental preconditioning algorithm. CIAL 1.1a differs from CIAL 1.1 in two ways. First, the time-consuming C library call `ieee_flags()`, which serves to set rounding directions and detecting floating-point exceptions, is replaced by assembly code. Second, similar to other CLP(Interval) languages, all intervals in CIAL 1.1a consist of closed bounds. We abandon the bound type calculation since the type information is insignificant in most cases and its calculation is costly.

We compare the CIAL 1.0 (Beta), CIAL 1.1, and CIAL 1.1a prototypes with BNR Prolog v3.1.0 [17], CLP(BNR) (or BNR Prolog v4.2.3) [3], Echidna Version 0.947 beta [18], ICL [11], and CIAL (Alpha)[12] [6] over randomly generated systems of linear equations of the form:

$$\begin{array}{lll} \mathbf{A}\vec{X} = \vec{b}, & \text{where} & \mathbf{A} = (a_{ij}), \vec{X} = (X_i), \vec{b} = (b_i) \text{ and } X_i \in [-10000, 10000] \\ & \text{for} & 1 \leq i, j \leq n. \end{array}$$

BNR Prolog runs on an Apple Mac II ($\sim$ 2 VAX MIPS with 5MB Ram) and the rest run on a SUN SPARCstation 10/30 ($\sim$ 49 VAX MIPS with 32MB Ram). Thus BNR Prolog's results should not be directly compared with others.

The results for each problem size n are the average of 3 different sets of test data and are summarized in table 1. The symbol "—" indicates a *failure*: either the system halts abruptly (trail/stack overflow) or fails to give solutions with width less than 1. The precision of the answers are set to 10 decimal places for CLP(BNR), ICL, and all CIAL prototypes, 5 decimal places for BNR Prolog, and the highest precision (`precision(30)` [19]) for Echidna. None of the interval-narrowing-based systems can handle randomly generated systems with rank greater than or equal to six[13].

Next we compare the three CIAL 1.x prototypes over larger scale systems and incremental execution. The randomly generated linear system of rank n is embedded in a program of $n+1$ predicates. The top level predicate `t/0` sets the initial bounds of all variables and calls predicate `p1/`n. Each subsequent predicate `p`i`/`n submits *one* constraint to the solver and calls `p(`$i+1$`)/`n. This call pattern exercises the incrementality of the linear solver to the fullest extent.

[12] CIAL (Alpha) uses generalized interval Gaussian elimination for linear constraint solving.

[13] This experiment does not imply that interval narrowing is incapable of solving *any* system of linear constraints with rank ≥ 6. Some large systems with special properties, e.g. strictly diagonal dominant, still can be solved even without using interval splitting.

	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
CIAL 1.x	0ms	0ms	0ms	$\leq$ 10ms	$\leq$ 10ms	$\leq$ 30ms
CIAL (Alpha)	10ms	40ms	80ms	190ms	380ms	600ms
BNR Prolog‡	6ms	89ms	17.36s	976.28s	9234.07s*	—
CLP(BNR)	10ms	37ms	1.69s	4.80s†	—	—
Echidna§	0.00s	0.00s	127.00s	—	—	—
ICL	11ms	28ms	7.45s	111.77s	210.02s*	—

Table 1: Average Computation Time in solving $\mathbf{A}\vec{X} = \vec{b}$

§ No timing built-ins: time is rounded to the nearest second.
‡ We are mainly interested in the rate of increase of BNR Prolog's results.
†The result is the average computation time of two sets of test data.
*The result is the computation time for only one set of test data.

Rank	CIAL 1.0 (Beta)	CIAL 1.1	Speedup	CIAL 1.1a	CLP($\mathcal{R}$)
50	85.06s	48.94s	1.74	22.50s	2.51s
60	171.73s	92.74s	1.85	45.04s	6.19s
70	371.58s	198.71s	1.87	76.38s	8.93s
80	544.25s	283.93s	1.92	122.47s	17.52s
90	838.93s	417.11s	2.01	180.68s	34.56s
100	1259.81s	607.91s	2.07	272.62s	39.07s

Table 2: Computation Time and Speedup

Table 2 shows the computation time for problems of size ranging from 50 to 100. As expected, CIAL 1.1 exhibits a near linear speedup over CIAL 1.0 (Beta) as the problem size grows. Although solutions given by CIAL 1.1 are slightly wider than those of CIAL 1.0 (Beta), solutions of CIAL 1.1 contain 8 significant digits of accuracy on average. The timing of CIAL 1.1a should not be compared against that of CIAL 1.0 (Beta) to measure speedup since the former involves no bound type calculation. We list the timing of CIAL 1.1a only to demonstrate the raw speed of CIAL, and the inefficiency of bound type calculation and the C library call `ieee_flags()`. The corresponding timing for CLP($\mathcal{R}$) for each problem is also given. CLP($\mathcal{R}$) responds in about 10~40 times faster than the CIAL's since interval computation is time-consuming. The solutions given by CLP($\mathcal{R}$), however, suffer from rounding errors. Many solutions given by CLP($\mathcal{R}$) fall out of the interval solutions returned by the CLP(Interval) systems in test at around the fourth or fifth decimal place. CIAL (Alpha) fails to returns answer intervals of useful width on any of the large linear systems.

5 Concluding Remarks

We have derived an incremental execution procedure of the preconditioned interval Gauss-Seidel method. Its correctness has also been established. The preliminary timing results have demonstrated the feasibility of using this method for efficient linear constraint solving in CLP(Interval) languages.

A number of questions remain to be investigated. On the theoretical side, it would be interesting to study the level of interval consistency attainable by the incremental preconditioned interval Gauss-Seidel method. We conjecture that our method should reach a consistency level falling between box consistency and hull consistency [2]. Concerning implementations, our CIAL prototypes have much to be desired. First, our implementation of CIAL 1.x is still rudimentary. Further optimizations, such as the techniques used for CLP($\mathcal{R}$), might be applicable to CIAL. Second, the current prototypes implement constraint solvers as independent modules separating from the Prolog engine. Communications between the solvers and the Prolog engine incur high overhead. The work of Lee and Lee [11] can be used as basis to integrate the interval constraint solving and the Prolog engine at the Warren Abstract Machine (WAM) level. We expect to improve the performance of CIAL 1.1 substantially in the near future.

Work is also in progress to adapt the interval Gauss-Seidel method for linear inequality constraint solving.

Acknowledgement

We are indebted to Spiro Michaylov, Roland Yap, and Tak-wai Lee for numerous discussions and comments. We thank the anonymous referees for their insightful suggestions. Access to CLP($\mathcal{R}$), BNR Prolog, CLP(BNR), Echidna, and ICL is gratefully acknowledged. This research is supported in part by RGC Earmarked Grant (CUHK 70/93E) from the University Grants Committee, and postgraduate studentships from the Chinese University of Hong Kong and the Fundação Macau.

References

[1] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1993.

[2] F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Logic Programming: Proceedings of the 1994 International Symposium*, Ithaca, USA, 1994.

[3] F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer and boolean constraints. *(to appear) Journal of Logic Programming.*

[4] X. Chen and D. Wang. On the optimal properties of the Krawczyk-type interval operator. *International Journal of Computer Mathematics*, 29(2-4):235–245, 1989.

[5] C.K. Chiu. Interval linear constraint solving in constraint logic programming. Master's thesis, Department of Computer Science, The Chinese University of Hong Kong, Shatin, Hong Kong, 1994.

[6] C.K. Chiu and J.H.M. Lee. Towards practical interval constraint solving in logic programming. In *Logic Programming: Proceedings of the 1994 International Symposium*, Ithaca, USA, 1994.

[7] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

[8] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, eighth edition, 1992.

[9] E.R. Hansen. Interval arithmetic in matrix computations. *SIAM Journal on Numerical Analysis*, 2:308–320, 1965.

[10] E.R. Hansen. A generalized interval arithmetic. In *Interval Mathematics*, pages 7–18, 1975.

[11] J.H.M. Lee and T.W. Lee. A WAM-based abstract machine for interval constraint logic programming. In *Proceedings of the Sixth IEEE International Conference on Tools with Artificial Intelligence*, New Orleans, USA, 1994.

[12] S. Michaylov. *Design and Implementation of Practical Constraint Logic Programming Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, U.S.A, August 1992.

[13] R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[14] A. Neumaier. Overestimation in linear interval equations. *SIAM Journal on Numerical Analysis*, 24(1):207–214, February 1987.

[15] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.

[16] W. Older and A. Vellino. Extending Prolog with constraint arithmetic on real intervals. In *Proceedings of the Canadian Conference on Computer & Electrical Engineering*, Ottawa, Canada, 1990.

[17] W. Older and A. Vellino. Constraint arithmetic on real intervals. In A. Colmerauer and F. Benhamou, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1992.

[18] G. Sidebottom and W.S. Havens. Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 8(4):601–623, 1992.

[19] S. Sidebottom, W. Havens, and S. Kindersley. *Echidna Constraint Reasoning System (Version 1): Programming Manual*. Expert Systems Laboratory, Simon Fraser University, British Columbia, Canada, 2.0 edition, 1992.

[20] P. Vasey. Qualified answers and their application to transformation. In *Proceedings of the Third International Logic Programming Conference*, pages 425–432, 1986.