, , 1–36 ()

Efficient Interval Linear Equality Solving in Constraint Logic Programming

C.K. CHIU AND J.H.M. LEE jlee@cse.cuhk.edu.hk
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong SAR, P.R. China

Editor:

Abstract. Existing interval constraint logic programming languages, such as BNR Prolog, work under the framework of interval narrowing and are deficient in solving systems of linear constraints over real numbers, which constitute an important class of problems in engineering and other applications. In this paper, we suggest to separate linear equality constraint solving from inequality and non-linear constraint solving. The implementation of an efficient interval linear constraint solver, which is based on the preconditioned interval Gauss-Seidel method, is proposed. We show how the solver can be adapted to incremental execution and incorporated into a constraint logic programming language already equipped with a non-linear solver based on interval narrowing. The two solvers share common interval variables, interact and cooperate in a round-robin fashion during computation, resulting in an efficient interval constraint arithmetic language CIAL. The CIAL prototypes, based on CLP($\mathcal{R}$), are constructed and compared favorably against several major interval constraint logic programming languages.

Keywords: Interval linear equalities, incremental execution, constraint logic programming

1. Introduction

The current status of Prolog arithmetic suffers from two deficiencies. First, the system predicate "is" [53] is functional in nature. It is incompatible with the relational paradigm of logic programming. Second, real numbers are approximated by floating-point numbers. Roundoff errors induced by floating-point arithmetic destroy the soundness [37] of computation. The advent of constraint logic programming [29] presents a solution to the first problem but the implementation of CLP languages, such as CLP($\mathcal{R}$) [30], are mostly based on floating-point arithmetic. The second problem remains.

The languages CAL [2] and RISC-CLP($\mathcal{R}$) [28] use symbolic algebraic methods to refrain from floating-point operations. Algebraic methods guarantee the soundness of numerical computation but they are time-consuming.

Previous efforts in the sub-symbolic camp, such as BNR Prolog [46], employ interval methods [41] and belong to the family of consistency techniques [38]. The main idea is to narrow the set of possible values of the variables of arbitrary real constraints using approximations of arc-consistency [9]. We collectively call these techniques *interval narrowing*. Interval narrowing has been shown to be applicable to critical path scheduling [46], X-ray diffraction crystallography [48], boolean constraint solving [8], and disjunctive constraint solving [8, 50]. However, interval narrowing can suffer from the "early quiescence" problem [18], in which case the

algorithm stops before reaching a good approximation of the set of possible solutions. This is usually a result of the interaction and coupling of the variables in the constraint system, and can happen to both linear and non-linear systems. This paper addresses the early quiescence problem for linear constraint systems.

For example, interval narrowing fails to solve such simple systems as "$\{X + Y = 5, X - Y = 6\}$." Cleary [15] proposes a form of case analysis technique [54], *domain splitting*, as a remedy. Domain splitting partitions an interval into two, visits one, and visits the other upon backtracking. This backtracking tree search is expensive to perform. Furthermore, interval narrowing may sometimes fail or take a long time to detect inconsistency of linear systems. Thus, interval narrowing is opted for improvement in terms of efficiency.

Our work is motivated by the inadequacy of interval narrowing for interval linear constraint solving. The goal is to design a *sound* and *efficient* interval linear constraint solving method for CLP languages. We suggest to separate linear equality constraint solving from inequality and non-linear constraint solving. This separation calls for an employment of two constraint solvers: a linear solver and a non-linear solver. Assuming the existence of an efficient non-linear solver, we aim to design and implement an interval linear solver in such a way that it (1) can handle interval linear systems efficiently and (2) can handle a mixture of interval linear and non-linear constraints by cooperating with an interval non-linear solver efficiently. We propose the use of the preconditioned interval Gauss-Seidel method as the backbone of an efficient linear equality solver. 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. 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 non-linear solver employs interval narrowing with splitting to solve inequalities and non-linear constraints. The two solvers interact and cooperate in a round-robin fashion during computation, resulting in a new efficient interval CLP system, CIAL (for *Constraint Interval Arithmetic Language*).

The paper is organized as follows. Section 2 reviews related work. Section 3 defines notations and contains preliminaries on interval arithmetic. Section 4 outlines the limitations of interval narrowing and interval splitting. Section 5 presents the implementation of an efficient linear solver and its correctness and complexity results. Section 6 describes the design of our new interval CLP system CIAL. The architecture of CIAL and the interaction among modules are explained. Section 7 describes the benchmarking results. Section 8 summarizes our contributions and sheds light on further work.

2. Related Work

Prolog III [16], CAL [2], and RISC-CLP($\mathcal{R}$) [28] use symbolic algebraic methods to solve arithmetic constraints. Prolog III[1] employs a simplex algorithm to handle

arithmetic over rational numbers. CAL computes over two domains: the real numbers and Boolean algebra with symbolic values. Constraints are solved by using the Buchberger algorithm for computing Gröbner bases [10]. RISC-CLP($\mathcal{R}$) deals with non-linear arithmetic constraints by using Gröbner basis and Partial Cylindrical Algebraic Decomposition [27, 11].

In the sub-symbolic camp, Cleary [15] introduces "logical arithmetic," a relational version of interval arithmetic, into Prolog. He describes distinct algorithms, one for each kind of 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. BNR Prolog [46] and its sequel CLP(BNR) [8] provide relational interval arithmetic in a way that is loosely based on Cleary's pre-publication idea, differing somewhat in particulars. Sidebottom and Havens [50] design and implement a version of relational interval arithmetic in the constraint reasoning system Echidna [24]. Based on hierarchical consistency techniques [39], Echidna can handle unions of disjoint intervals. Sidebottom [51] describes the use of projection constraints for compiling and optimizing constraint propagation in the numeric and Boolean domains. He shows that all the constraints available in CLP(BNR) can be directly expressed by projection constraints without applying constraint decomposition. Also, the user can program different constraint propagation methods for different constraints. Lhomme [36] analyzes the complexity of consistency techniques for numeric CSP's and proposes partial consistency techniques, whose complexities can be tuned by adjusting the bound width of the resulting intervals. Lee and van Emden [34, 35] generalize Cleary's algorithms for narrowing intervals constrained by any relations p on $I(\mathbb{R})^n$. They also show how the generalized algorithm can be incorporated in CLP($\mathcal{R}$) [30] and CHIP [19] in such a way that the languages' logical semantics is preserved. Lee and Lee [33] propose an integration of constraint interval arithmetic into logic programming at the Warren Abstract Machine (WAM) [3] level. Benhamou *et al* [7] and Van Hentenryck *et al* [56] replace the usual interval narrowing operator of previous interval CLP languages by an operator based on the interval Newton method to speed up non-linear constraint solving. This work on interval Newton leads to Numerica [57, 55], a modeling language for stating and solving global optimization problem. Numerica allows the expression of problems in a notation close to the way that they are stated in textbooks or scientific papers, and provides guarantees about correctness, convergence, and completeness. Chiu and Lee [13] use generalized interval Gaussian elimination to improve the efficiency of interval linear equality constraint solving.

3. Notations and Preliminaries

Constraints in CIAL are over real numbers. An interval is represented by an appropriate pair of inequality constraints bounding the value of a variable occurring in a constraint or a logical-valued expression, which represents an unknown real number. We denote this kind of variable by such typewriter-like upper case letters as $\mathtt{X}, \mathtt{Y}$ and $\mathtt{Z}$. For example, $\{\mathtt{X} > 3, \mathtt{X} \leq 6\}$ denotes the relation $\mathtt{X} \in (3, 6]$. Mathematical

interval variables (or constants) referring to non-empty floating-point intervals are denoted by upper (or lower) case letters with superscript I, while 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. We overload the $\sum$ symbol to denote summation in the real, floating-point, and interval domains. The exact meaning of the symbol can be inferred from the context of where the symbol appears.

The rest of this section provides the theoretical background to this paper. The basics of interval arithmetic are presented. We then describe the syntax and semantics of ICLP($\mathcal{R}$) language [34], which is the language basis of CIAL.

3.1. Interval Arithmetic Notations

The books [41, 42, 4] provide good introduction to interval analysis. Let $\mathbb{R}$ be the set of real numbers and $\mathbb{F}$ a set of floating-point numbers, which include the element 0. Mathematically, a *real interval* is a segment, possibly infinite, of the real line and can be defined by an ordered pair of real numbers $a \leq b$, where a is the lower bound and b is the upper bound. For those intervals without upper bound or lower bound, we use the symbols $-\infty$ and $+\infty$ as bounds respectively. Note that $-\infty$ and $+\infty$ can only be used with open bounds. An interval is represented by the usual mathematical notation, such as $[1, 10)$ which denotes the set $\{x \mid 1 \leq x < 10\}$. We differentiate between real intervals and floating-point intervals. The bounds of the elements of the former are real numbers; while the bounds of elements of the latter are restricted to floating-point numbers.

The *set of real intervals* $I(\mathbb{R})$, induced by $\mathbb{R}$, is defined by

$$\begin{aligned} I(\mathbb{R}) \;=\; & \{(a,b] \mid a \in \mathbb{R} \cup \{-\infty\}, b \in \mathbb{R}\} \cup \{[a,b) \mid a \in \mathbb{R}, b \in \mathbb{R} \cup \{+\infty\}\} \cup \\ & \{[a,b] \mid a, b \in \mathbb{R}\} \cup \{(a,b) \mid a \in \mathbb{R} \cup \{-\infty\}, b \in \mathbb{R} \cup \{+\infty\}\}. \end{aligned}$$

The set of floating-point intervals $I(\mathbb{F})$, induced by $\mathbb{F}$, is defined similarly. We can verify that $I(\mathbb{F}) \subset I(\mathbb{R})$.

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 that B^I does not contain 0.

We formally express *outward-rounding* by $\xi : I(\mathbb{R}) \to I(\mathbb{F})$. If J^I is a non-empty real interval,

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

The outward-rounding function gives the tightest floating-point interval containing J^I.

Cleary [15] introduces "logical arithmetic," which is a relational version of interval arithmetic, by defining distinct primitive arithmetic constraints over intervals. The values of intervals that do not satisfy a constraint are eliminated by applying interval reduction [32] on the constraint. Interval reduction applies only to individual constraints. In practice, there is usually more than one constraint in a relational interval arithmetic system, resulting in a *constraint network*. The constraints interact with one another by sharing intervals. A relaxation algorithm [15], which is similar to the arc-consistency algorithm AC-3 [38], is used to coordinate the execution of interval reduction to narrow the intervals in a constraint network. An interval constraint is *stable* if all bounds of its variables remain unchanged after interval reduction is applied on the constraint; otherwise the constraint is *active*. A network is *stable* if all constraints in the network are stable. Lee and van Emden [35] show that the relaxation algorithm always terminates, in which case the constraint network is either shown to be inconsistent, or reduced to stability. By *interval narrowing*, we refer to the relaxation algorithm plus interval reduction operators defined for each type of primitive constraint in the network.

3.2. Syntax and Semantics

The language framework of CIAL is based on ICLP($\mathcal{R}$) [34]. We give the syntax and semantics of ICLP($\mathcal{R}$) in the following.

ICLP($\mathcal{R}$) and CLP($\mathcal{R}$) share the same syntax and declarative semantics [29, 30]. An interval constraint in CIAL is expressed as,

$$X_1 \in I_1^I, \cdots, X_n \in I_n^I,\ p(X_1, \ldots, X_n),$$

where p is a relation on $\mathbb{R}^n$ and $X_i \in I_i^I$ is an appropriate pair of inequalities. The operational semantics is based on the generalized $\mathcal{M}_\chi$ derivation [32], which is shown in the following.

Let P be an interval CLP program and G_p be a goal in the form `?-` $\vec{\Sigma}, \vec{\Theta}, \vec{\Delta}$, where $\vec{\Sigma}$ is a set of stable constraints, $\vec{\Theta}$ is a set of active constraints, $\vec{\Delta}$ is a set of atoms[2], and "`?-`" is a symbol to indicate a goal or query in constraint logic programming. Initially $\vec{\Sigma}$ is empty. Without loss of generality, an initial goal G_0 is always of the form `?-` G, where G is an atom. A derivation step that reduces a goal G_p to another G_{p+1} follows:

- $\gamma \in \vec{\Delta}$ and the program P contains a rule[3] H `:-` $\vec{\Theta'}, \vec{\Delta'}$, such that the head atom H can be unified with γ using substitution θ, i.e. $H\theta = \gamma\theta$. G' is

  `?-` $((\vec{\Sigma} \bigcup \vec{\Theta}), \vec{\Theta'}, (\vec{\Delta} \setminus \gamma) \bigcup \vec{\Delta'})\theta$.

- G_{p+1} is G' with $(\vec{\Sigma} \bigcup \vec{\Theta})\theta$ replaced by $F_{nf}((\vec{\Sigma} \bigcup \vec{\Theta})\theta)$, where F_{nf} is a *normal function* that maps from set of constraints to set of constraints in such a way that

$$P \models_{\mathcal{M}_\chi} \exists((\vec{\Sigma} \bigcup \vec{\Theta})\theta) \Leftrightarrow P \models_{\mathcal{M}_\chi} \exists(F_{nf}((\vec{\Sigma} \bigcup \vec{\Theta})\theta))$$

where $P \models_{\mathcal{M}_\chi} F$ means that the formula F is a logical consequence of P.

THEOREM 1 *[32] If C' is obtained from C using interval reduction on p, where C is $X_1 \in I_1^I, \ldots, X_n \in I_n^I$, $p(X_1, \ldots, X_n)$ and C' is $X_1 \in I_1^{I'}, \ldots, X_n \in I_n^{I'}$, $p(X_1, \ldots, X_n)$, then $\models_{\mathcal{M}_\chi} \exists(C) \Leftrightarrow \models_{\mathcal{M}_\chi} \exists(C')$.*

Theorem 1 shows that interval reduction transforms an interval constraint into another one with the same solution space. Interval narrowing, which performs interval reduction repeatedly on interval constraints in a constraint network, is therefore a normal-form function.

A generalized $\mathcal{M}_\chi$ derivation is a sequence of goals, possibly infinite. A derivation is *successful* if it is finite and the last goal is empty; *finitely-failed* if it is finite but the last goal has one or more atoms. The generalized $\mathcal{M}_\chi$ derivation ends with a *floundered goal* if the last goal has one or more stable constraints. Floundered goal gives "incomplete" solutions and should be interpreted as conditional answers [58]. Suppose a non-empty goal `?-` $\vec{\Sigma}, \vec{\Theta}, \vec{\Delta}$ is derived from an initial goal `?-` G and θ is a composition of all the substitutions. The rule (G`:-` $\vec{\Sigma}, \vec{\Theta}, \vec{\Delta})\theta$ is a *conditional answer* to the original goal.

4. Limitations of Interval Narrowing

Interval narrowing with splitting is a common constraint solving technique used in interval constraint logic programming languages [35, 33, 8, 46]. Recent results [12] reveal that this technique is deficient in solving general systems of linear constraints. We summarize the results here. Readers may refer to [12] for detailed analysis.

Interval narrowing can be classified as a fixed-point iterative method. Whether it converges to reasonable sharp solutions depends highly on the initial bounds of variables and the form of interval constraints. An obvious example is that interval narrowing fails to narrow any variables in the simple system $\{X = Y, X = -Y\}$ with initial bounds $X, Y \in [-50, 50]$. Interval splitting [15] is a divide-and-conquer algorithm for obtaining sharper interval solutions. It partitions an interval into two, visits one, and visits the other upon backtracking. The efficiency of this technique depends on the interval subdivision method and search strategy. In general, interval narrowing with splitting is both time and storage consuming. Thus it is impractical to use the combined technique on general system of interval linear constraint solving. Chiu [12] justifies this claim by experiments on solving a set of randomly generated systems of linear constraints of the form

$$\begin{aligned} \mathbf{A}\vec{X} = \vec{b}, \text{ where } & \mathbf{A} = (a_{ij}), \vec{X} = (X_i), \vec{b} = (b_i), a_{ij} \neq 0, b_i \neq 0, \\ & \text{and } X_i \in [-10000, 10000] \\ \text{for } & 1 \leq i, j \leq n \end{aligned}$$

with BNR Prolog [47], CLP(BNR) [8], Echidna [50] and ICL [33]. All the above interval-narrowing-based systems fail to solve (either the program halts abruptly

with trail/stack overflow or fails to give solutions with width less than 1) linear systems with rank greater than 5. The performance of interval narrowing with splitting has not been improved significantly when 20% to 40% of the coefficients are replaced by zeros randomly. Even for sparse linear systems with 60% zero-coefficients, only those of rank less than 11 can be handled.

Another deficiency of interval narrowing with splitting is its inability to detect inconsistency. As stated in section 3.2, answers obtained from interval narrowing should be regarded as conditional. A set of inconsistent constraints can be narrowed to become stable without inconsistency being found. A simple example is

$$\begin{cases} A+1 &= D & (C_1) \\ A+B &= D & (C_2) \\ A, D &\in [0, \infty) \\ B &\in (-\infty, 0]. \end{cases}$$

Equations (C_1) and (C_2) imply $B = 1$, which contradicts the fourth constraint $B \in (-\infty, 0]$. This inconsistency, however, cannot be detected by interval narrowing even when interval splitting is applied [12]. Besides the incompleteness of inconsistency detection, this example exhibits another important shortcoming of interval narrowing. Both the lower bounds of variable A and D are narrowed towards $+\infty$ in the narrowing process. However, since the lower bounds move in increment of 1 in each narrowing step, interval narrowing may take a long time to stablize.

Benhamou *et al* [7] and Van Hentenryck *et al* [56] show an improvement on interval narrowing. The results are implemented in the `Newton` language. Chiu [12] shows that their improvement applies only to interval non-linear constraint solving. With linear constraints, the `Newton` procedure [7] degenerates to interval narrowing.

5. An Efficient Interval Linear Solver

Motivated by the deficiencies of interval narrowing, our goal is to design a good linear solver, which should satisfy the following criteria:

1. The linear solver must be amenable to efficient *incremental execution*. The complexity of adding and solving a new constraint should be affected more by the form of the new constraint, rather than of the constraints already collected in the linear solver [40].

2. Linear constraint solving in the linear solver must be substantially more *efficient* than interval narrowing.

3. Solutions given by the solver must be *sound* and *accurate*. The former criterion implies that the real solutions should always fall into the answer intervals. To satisfy the latter, the widths of answer intervals should be less than a reasonable small value.

We propose the use of the preconditioned interval Gauss-Seidel method as the backbone of such a linear constraint solver. 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. 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.

5.1. 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. The interval Gauss-Seidel method is one being widely-used in interval computation [44].

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

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

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

$$X_i^I \leftarrow ((b_i^I \ominus \sum_{j=1,j\neq i}^{n}(a_{ij}^I \otimes X_j^I)) \oslash a_{ii}^I) \bigcap X_i^I$$

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 iteration 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 [5] is suggested to be

$$X_i^{I(k)} \leftarrow ((b_i^I \ominus \sum_{j=1}^{j=i-1}(a_{ij}^I \otimes X_j^{I(k)}) \ominus \sum_{j=i+1}^{j=n}(a_{ij}^I \otimes X_j^{I(k-1)})) \oslash a_{ii}^I) \bigcap X_i^{I(k-1)}.$$

The superscript $(k-1)$ of $X_i^{I(k-1)}$ indicates that the variable is obtained in the $(k-1)$-st iteration cycle. The interval Gauss-Seidel method terminates when all variables remain unchanged after an iteration or when the difference between the new and the last computed values 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 1. [44] A sequence of intervals *converges* iff both the lower and upper bounds converge.

Definition 2. [44] 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 3. [44] 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 diagonally dominant* if

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

LEMMA 1 *[44] The interval Gauss-Seidel method is guaranteed to converge to the hull of the solution set of a linear system if the coefficient matrix of the linear system is strictly diagonally dominant.*

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.

5.2. Preconditioning

As stated in Lemma 1, the interval Gauss-Seidel method always converges to the hull of the solution set on a system with strictly diagonally 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 diagonally 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 [22] suggests the inverse mid-point matrix as preconditioner. Let $\check{\mathbf{A}}$ denote the mid-point matrix of $\mathbf{A^I}$. We define

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

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

Since preconditioning involves many interval multiplications, small errors will be introduced due to interval dependency [44, pages 118–119] and outward-rounding.

A preconditioned system usually has slightly wider solutions than the original system and these additional pseudo-solutions are called overestimation [43]. Overestimation destroys the completeness of inconsistency detection in the interval Gauss-Seidel method, since an inconsistent system of constraints may become consistent after preconditioning. Readers may refer to [43, 44] for detailed analysis.

5.3. 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 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.

5.3.1. Incremental Update of Preconditioner We adopt the inverse mid-point matrix preconditioner, as stated in Section 5.2. Assume that we have a collection of m interval linear equalities of n 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 a computer in general. We can simply round them to their nearest floating-point numbers. 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 language. The linear system in the solver does 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 a heuristic (to be discussed in Section 5.3.2) 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

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

The preconditioner $\mathbf{P}$ is computed (to be explained in the next paragraph and described in details in Algorithm 1) by row reducing the combined matrix $[\check{\mathbf{A}}|\mathbf{J}]$ until 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. Note that $\mathbf{P}$ is an $m \times m$ matrix. 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($\mathcal{R}$) [40]. 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. Step 3 of the algorithm chooses the largest of $x_{r+1,l}$ for $r+2 \leq l \leq n$

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

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

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. (Pivoting) We interchange the $(r+1)$-st column of $\mathbf{X}$ with a column chosen from the $(r+1)$-st to n-th columns, say the s-th, such that $x_{r+1,r+1} \geq x_{r+1,l}$ for $r+2 \leq l \leq n$.

   We also have to interchange the $(n+r+1)$-st column with the $(n+s)$-th column accordingly. All interchanges are recorded to guarantee that their associated variables in the constraints can be identified.

4. (Row Normalization) We divide all the elements in the $(r+1)$-st row by a suitable value such that $x_{r+1,r+1}$ becomes 1.

5. (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

and performs pivoting to make $x_{r+1,r+1}$ as large as possible. This pivoting step is necessary since $x_{r+1,r+1}$ is used as the divisor in the row normalization operation in Step 4. A larger normalizer can help suppress the magnitude of normalized entries in the $(r+1)$-st row. The updated $(r+1) \times (r+1)$ preconditioner resides in the $(n+1)$-st to $(n+r+1)$-st columns of the IUPZ matrix $\mathbf{X}$.

An alternative to Algorithm 1 is to use linear programming to compute the width optimal preconditioner [31, chapter 3].

5.3.2. 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 the 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 has 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 execution of the preconditioned system will terminate prematurely and fail to return sharp intervals that correspond to the actual answers of the system. In the worst case, the initial intervals of the variables are returned as answers un-narrowed. It is important to note that the guaranteed inclusion property of interval methods still holds although the answers returned may be of no practical use.

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 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 our implementation 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 given by our system are interpreted as conditional answers [58, 12].

LEMMA 2 *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.

The detection of an inconsistent or redundant constraint takes $(2n - i + 1)ic_1$ operations since it involves only forward substitution. The *worst* case scenario occurs when all inconsistent/redundant constraints arrive after $(n-1)$ independent constraints have been added. This way the function $f(i) = (2n-i+1)ic_1$ where $i \in \{1, \ldots, n-1\}$ is maximized. 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

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

■

5.3.3. 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. Extra constraints should have been moved to another solver in the phase of inconsistency and redundancy detection. The application of preconditioning involves multiplication of a point preconditioner matrix and an interval coefficient matrix, which is of $O(r^2 n)$ complexity[4], 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 $\mathbf{O}' \otimes \mathbf{A'^I} \otimes \vec{X^I} = \mathbf{O}' \otimes \vec{b'^I}$ 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* $\mathbf{O}' \otimes \mathbf{A'^I}$ *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 $\mathbf{A'^I}$ 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 matrix $\mathbf{R^I}$ can be calculated by the following equations:

$$\begin{aligned} \mathbf{R_1^I} &= \mathbf{O} \otimes \mathbf{A_1^I} \oplus \mathbf{L} \otimes \mathbf{A_3^I} & \mathbf{R_2^I} &= \mathbf{O} \otimes \mathbf{A_2^I} \oplus \mathbf{L} \otimes \mathbf{A_4^I} \\ \mathbf{R_3^I} &= \mathbf{M} \otimes \mathbf{A_1^I} \oplus \mathbf{N} \otimes \mathbf{A_3^I} & \mathbf{R_4^I} &= \mathbf{M} \otimes \mathbf{A_2^I} \oplus \mathbf{N} \otimes \mathbf{A_4^I}. \end{aligned} \tag{2}$$

All terms in (2), except $\mathbf{O} \otimes \mathbf{A_1^I}$ and $\mathbf{O} \otimes \mathbf{A_2^I}$, can be calculated in $O(r^2)$ or $O(r(n-r))$ time. We concentrate on the computation of $\mathbf{O} \otimes \mathbf{A_1^I}$ and $\mathbf{O} \otimes \mathbf{A_2^I}$.

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.

$$\left[\begin{array}{c:c} \mathbf{O}_{r\times r} & \mathbf{L}_{r\times 1} \\ \hdashline \mathbf{M}_{1\times r} & \mathbf{N}_{1\times 1} \end{array}\right] \otimes \left[\begin{array}{c:c} \mathbf{A^I}_{1_{r\times r}} & \mathbf{A^I}_{2_{r\times(n-r)}} \\ \hdashline \mathbf{A^I}_{3_{1\times r}} & \mathbf{A^I}_{4_{1\times(n-r)}} \end{array}\right] = \left[\begin{array}{c:c} \mathbf{R^I}_{1_{r\times r}} & \mathbf{R^I}_{2_{r\times(n-r)}} \\ \hdashline \mathbf{R^I}_{3_{1\times r}} & \mathbf{R^I}_{4_{1\times(n-r)}} \end{array}\right]$$

$$\mathbf{P}_{(r+1)\times(r+1)} \qquad \mathbf{A^I}_{(r+1)\times n} \qquad \mathbf{P}_{(r+1)\times(r+1)} \otimes \mathbf{A^I}_{(r+1)\times n}$$

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

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

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

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

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.

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

It follows that $\mathbf{O} \otimes \mathbf{A_1^I}$ and $\mathbf{O} \otimes \mathbf{A_2^I}$ can be approximated[5] by

$$\mathbf{O} \otimes \mathbf{A_1^I} \subseteq \mathbf{O}' \otimes \mathbf{A_1^I} \ominus \begin{pmatrix} u_1 \\ u_2 \\ \dots \\ u_r \end{pmatrix} \otimes ((t_1\ t_2\ \dots\ t_r) \otimes \mathbf{A_1^I}) \tag{5}$$

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

using the *subdistributivity*[6] and *associativity*[7] [41] 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 computation 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:

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

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 first define the notion of *inward-rounding*: if J^I is a non-empty real interval, the *inward-rounding function* $\eta : I(\mathbb{R}) \to I(\mathbb{F})$ is defined as

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

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 interval subtraction and multiplication respectively.

LEMMA 3

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

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

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

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

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

Similarly, we can show

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

■

By Lemma 3, $\mathbf{P'}$ 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 $(u_1, u_2, \dots, u_r)^T$.

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 4 *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 [41], we know that $\vec{x'}^I \subseteq \vec{x_2}^I$. It follows that $\vec{x_1}^I \subseteq \vec{x_2}^I$. ■

THEOREM 2 *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 2, 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 and 4, the incremental preconditioning algorithm preserves all solutions of an interval linear system. ■

5.3.4. Interval Propagation The incremental preconditioning algorithm performs only constraint transformations. The domains of the 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

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

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

$$X_k^I \leftarrow ((b_i^I \ominus \sum_{j=1, j\neq k}^{n} (a_{ij}^I \otimes X_j^I)) \oslash a_{ik}^I) \bigcap X_k^I \quad \text{for } k = i \text{ and } m+1,\ldots,n. \tag{8}$$

This step is effective since the IUPZ matrix is diagonally dominant only in the first m columns. Thus the entry a_{ik}^I for $k = m+1,\ldots,n$ is not necessarily small as compared to the numerator (although a_{ik}^I is typically of tiny magnitude for $k = 1,\ldots,m$ and $k \neq i$).

The invocation of constraints in the linear constraint solver may not be organized in a round-robin fashion. We discuss the details in Section 6.4.

6. Design of CIAL

We extend ICLP($\mathcal{R}$) [34], which is an extension of CLP($\mathcal{R}$) with interval narrowing, with our proposed efficient linear constraint solver, resulting in a new interval constraint logic programming system, CIAL (for *Constraint Interval Arithmetic Language*). The syntax and semantics of CIAL are almost identical to those of ICLP($\mathcal{R}$), except that the relational symbol "=" is replaced by "=:=". In this section, we describe the modules of CIAL and explain how they interact. Unification between interval variables and other types of data, and decomposition of interval constraints will also be discussed.

6.1. The CIAL Architecture

Figure 2 gives an overview of the CIAL architecture. The *input* and the *engine*

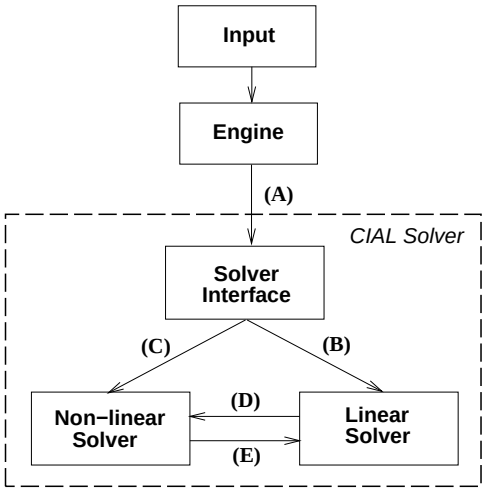

Figure 2. An CIAL Architecture

components are adaptation of a Prolog interpreter. Their functions include unification, goal reduction, and delivery of constraints collected at each derivation step to the *solver interface*. The interface in turn decomposes and distributes the constraints to the *linear solver* and the *non-linear solver* accordingly. In the following, we describe each component of the architecture and the interaction between the two solvers in more detail.

6.2. The Inference Engine

The structure of the engine resembles that of a standard structure-sharing Prolog interpreter [3]. Equations between Prolog terms are handled by a standard unification algorithm. Since constraints in CIAL are over *real numbers*, variables occurring in constraints denote *unknown real numbers*. We refer to those variables as interval variables. The introduction of interval variables calls for an extension of the standard unification algorithm.

6.2.1. Interval Variables A variable `X` in a constraint becomes an *interval variable* when it is involved in such simple inequality constraints as "$\mathtt{X} > 3, \mathtt{X} \leq 6$," or such equality constraints as "$\mathtt{X} + 2 * \mathtt{Y} =:= \mathtt{Z}, \mathtt{X} * \mathtt{X} =:= \mathtt{Y}$." In the first example, we say that the interval $(3, 6]$ is *associated* with the variable `X`. Semantically, an

interval variable is an ordinary variable occurring in a constraint or a logical-valued expression. We distinguish interval variables from variables occurring in constraints or logical-valued expressions purely for implementation efficiency.

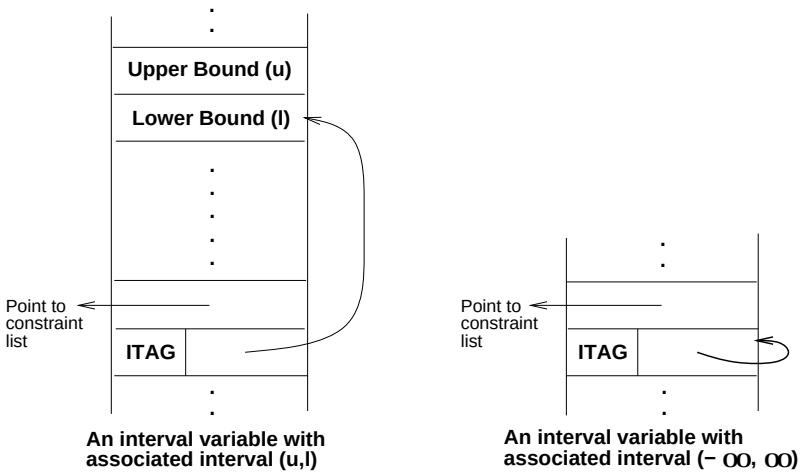

Figure 3. The Heap Representations of Interval Variables

Resembling domain variables in finite-domain languages [54], e.g. CHIP [1, 19], an interval variable is represented as a variable with an associated interval. Its heap representation is shown in Figure 3, where ITAG is a new tag introduced for interval variables. Each interval variable in CIAL keeps a list of constraints in which the variable appears. When an interval variable is narrowed, we can locate and *wake up* its related constraints efficiently via the constraint list. By waking up a constraint, we mean moving the constraint from the passive list to the active list. This list is important since interval variables are modified often during computation.

6.2.2. Extended Unification Algorithm Additional binding mechanisms are defined for unification between:

- *an interval variable and a free variable*
  We simply bind the free variable to the interval variable. No constraint solver will be invoked.

- *an interval variable and an interval variable*
  To unify two interval variables `X` and `Y`, we compute the intersection of their associated intervals. If the intersection J^I is non-empty, we choose one of `X` and `Y` (for efficiency reasons, we choose the variable which does not require trailing, if possible), say `X`, and bind it to the other, `Y` in this case. Then we replace the

associated interval of the chosen variable `X` by the intersection J^I. Otherwise, failure is reported.

- *an interval variable and a number*
  We treat a number as an interval variable, the associated interval of which has the number as closed upper and lower bounds. Thus, unification between a number and an interval variable can be performed in the same way as unification between two interval variables.

- *an interval variable and other terms*
  Failure is reported.

When two interval variables are unified successfully, their constraint lists are merged and all related constraints are waked up.

6.3. The Solver Interface and Constraint Decomposition

The design of the CIAL solver interface is similar to that of CLP($\mathcal{R}$) [30]. The solver interface is called from the inference engine whenever a constraint contains an arithmetic term. If the input constraint contains any number that cannot be represented exactly as a floating-point number, the number will be first outward-rounded to an interval. The constraint is then simplified by evaluating the arithmetic expression. For example, the constraint "`3+9-2*X=:=Y+4*X`" is simplified to "`12=:=Y+6*X`." If the simplified constraint is an equality with only one variable, it is resolved in the interface according to the extended unification algorithm. In all other cases, the input constraint will be decomposed and then distributed to the linear and the non-linear constraint solver accordingly.

CLP($\mathcal{R}$) differentiates between directly solvable constraints and hard constraints [40]. The former are solved by either Gaussian elimination (for linear equalities) or Simplex method (for linear inequalities) once they are collected, while the latter are delayed from consideration until they become linear. We do otherwise in CIAL. We do not delay any constraint. Once constraints are collected in a derivation step, they will be narrowed in either the linear or the non-linear solvers. To maximize the efficiency of constraint solving, we classify constraints into three categories.

An interval is *non-narrowable* if its width is less than a user-defined value or if it cannot be further split in the underlying floating-point system (i.e. when the lower and upper bounds of the interval are "adjacent" in the floating-point line). Otherwise the interval is *narrowable*. A variable is *non-narrowable* if its associated interval is non-narrowable. Otherwise, the variable is *narrowable*. A *constant* is either a floating-point number or a non-narrowable variable. A narrowable variable is a *linear term*. The multiplication of several terms is also a *linear term* if it involves only constants and at most one linear term. Otherwise, the product is a *non-linear term*. A *linear constraint* contains only summation of linear terms and constants, while a *non-linear constraint* contains only summation of non-linear terms and constants. A constraint is *mixed* if it contains both linear and non-linear terms.

In CIAL, a linear constraint goes directly to the linear constraint solver without being pre-processed. A non-linear constraint is first partitioned into a set of convex primitives, as described in [15], and then delivered to the non-linear solver. For a mixed constraint, we decompose it into a linear constraint and a set of non-linear constraints. The resultant constraints are handled as ordinary linear or non-linear constraints. The decomposition procedure is shown in Algorithm 2.

---

1. We *linearize* a mixed constraint by replacing all non-linear terms, each by a temporary variable. Each of the non-linear terms and its corresponding temporary variable are associated by the relational symbol "=:=", resulting in a new non-linear constraint.

2. The linearized constraint is in the form,

$$f(\mathtt{X_1}, \cdots, \mathtt{X_n}) =:= \mathtt{c} + \sum_{k=1}^{j} (s_k \times \mathtt{T_k}),$$

   where $f(\mathtt{X_1}, \cdots, \mathtt{X_n})$ is a linear arithmetic term involving only program/query variables, $\mathtt{X_i}$'s are program/query variables, $\mathtt{c}$ is a constant, $\mathtt{T_k}$'s are the temporary variables introduced to replace non-linear terms, and $s_k = 1$ or -1 for $k = 1, 2, \cdots, j$.

3. The linearized constraint is then partitioned into two by separating the program/query variables from the temporary variables,

$$f(\mathtt{X_1}, \cdots, \mathtt{X_n}) =:= \mathtt{c} + \mathtt{T_0} \quad \text{and} \quad \mathtt{T_0} =:= \sum_{k=1}^{j} (s_k \times \mathtt{T_k}).$$

4. We pass the constraint $f(\mathtt{X_1}, \cdots, \mathtt{X_n}) =:= \mathtt{c} + \mathtt{T_0}$ to the linear solver. The constraint $\mathtt{T_0} =:= \sum_{k=1}^{j} (s_k \times \mathtt{T_k})$ and all non-linear constraints are partitioned into primitives and they are delivered to the non-linear solver.

Algorithm 2. Constraint Decomposition Procedure

To improve the efficiency of linear constraint solving, we pass the linear constraint $\mathtt{T_0} =:= \sum_{k=1}^{j} (s_k \times \mathtt{T_k})$ to the non-linear solver instead of the linear solver. Interval linear constraint solving usually involves variable elimination [20, 23, 44], which is a time consuming process. If we deliver the constraint $\mathtt{T_0} =:= \sum_{k=1}^{j} (s_k \times \mathtt{T_k})$ to the linear solver, the temporary variables $\mathtt{T_k}$, for $k = 1, 2, \ldots, j$, are unique there and they can never be eliminated. Thus, such a delivery does not help to give sharper results, but only increases the number of operations unnecessarily.

We illustrate our constraint decomposition procedure by considering the following query,

$$?- \quad 3*X+5*Y-(X+Y)*(6-Z)+Z*Z =:= 10,\\ X+Y =:= 20, X*Y*Z =:= 12.$$

The underlined constraints are generated during decomposition.

1. The first constraint is linearized and we have

$$\underline{3*X+5*Y-T1+T2 =:= 10, T1 =:= (X+Y)*(6-Z), T2 =:= Z*Z},\\ X+Y =:= 20, X*Y*Z =:= 12.$$

   Note that the term $(X+Y)*(6-Z)$ should not be further translated into

$$6*X+6*Y-Z*(X+Y)$$

   which introduces one more occurrence of the variables X and Y. Such a translation aggravates the effect of the variable dependency problem [21].

2. We minimize the number of temporary variables in the linearized constraint,

$$3*X+5*Y-T1+T2 =:= 10,$$

   by partitioning it into two,

$$\underline{3*X+5*Y+T0 =:= 10, T0 =:= -T1+T2}, T1 =:= (X+Y)*(6-Z),\\ T2 =:= Z*Z, X+Y =:= 20, X*Y*Z =:= 12.$$

3. We further decompose the two non-linear constraints,

$$T1 =:= (X+Y)*(6-Z), X*Y*Z =:= 12,$$

   into a set of convex primitive constraints.

$$3*X+5*Y+T0 =:= 10, T0 =:= -T1+T2, \underline{T4 =:= X+Y, T5 =:= 6-Z},\\ \underline{T1 =:= T4*T5}, T2 =:= Z*Z, X+Y =:= 20, \underline{X*Y =:= T3, T3*Z =:= 12}.$$

4. The linear constraints,

$$3*X+5*Y+T0 =:= 10, X+Y =:= 20,$$

   are passed to the linear constraint solver, while the others,

$$T0 =:= -T1+T2, T4 =:= X+Y, T5 =:= 6-Z, T1 =:= T4*T5,\\ T2 =:= Z*Z, X*Y =:= T3, T3*Z =:= 12,$$

   are delivered to the non-linear constraint solver.

6.4. The Linear and the Non-linear Solvers

In traditional interval constraint logic programming languages, all interval constraints are solved under a uniform framework, interval narrowing. To improve the efficiency of interval constraint solving, we separate linear equality constraint solving from inequality and non-linear constraint solving in CIAL.

CIAL consists of two constraint solvers, a linear constraint solver and a non-linear constraint solver. The former is responsible only for linear equality constraints. Non-linear constraints and inequalities belong to the latter. The linear constraint solver is based on the incremental preconditioned interval Gauss-Seidel method; while the non-linear constraint solver employs interval narrowing with splitting as the constraint solving technique.

The employment of more than one solver in CIAL calls for an interaction scheme. We explain in Algorithm 3 how the two solvers cooperate in one constraint solving step. Letters in parentheses refer to the labels in Figure 2. Steps 2 and 5-6 correspond to the preconditioning phase and the interval propagation phase respectively in the linear constraint solving.

A non-linear primitive constraint is sent to the linear solver only when the constraint becomes linear and it does not contain any temporary variable. Primitive constraints with temporary variables always stay in the non-linear solver since they usually cannot help to eliminate any variable in the constraints in the linear solver.

THEOREM 3 *The constraint solving step in Algorithm 3 always terminates. The system is either inconsistent or stable.*

Proof: The input constraints are finite. It implies that the invocations of the constraint decomposition (Step 1) and the preconditioning (Step 2) procedures are finite. Algorithm 3 halts either when the value of a variable becomes empty or both the lists LA and NA become empty. In the former case, the system is inconsistent. For the latter case, we observe that the size of the lists LA and NA decreases after each iteration unless variables are narrowed. However, the precision of a floating-point system is finite and no variables can be narrowed indefinitely. Therefore, both the lists LA and NA must become empty after a finite number of iterations. All constraints become stable. Thus the system is stable.

7. Benchmarkings

In order to demonstrate the feasibility of our proposal, we have constructed several CIAL prototypes using the C programming language. Since CIAL has much in common with CLP($\mathcal{R}$) on the surface, we decide to use CLP($\mathcal{R}$) as the backbone of our implementation and try to adopt as much original CLP($\mathcal{R}$) code as possible. It turns out that only the Prolog engine part of CLP($\mathcal{R}$) can be re-used in our implementation. The solver interface and the two solvers are implemented from scratch. We also modify the unification algorithm of the Prolog engine to cope with unification between interval variables and other terms. All the implementations are based on CLP($\mathcal{R}$) Version 1.2.

Let LA and NA be two *active lists* which contain active constraints in the linear and the non-linear solvers respectively.

1. A new interval constraint will be resolved in the solver interface if possible. Otherwise, if the constraint is non-linear or mixed, it is decomposed into a conjunction of primitive constraints, or a linear constraint and a conjunction of primitive constraints respectively `(A)`.

2. The linear constraint is sent to the linear solver. The preconditioner and the preconditioned system are updated incrementally to include the new linear constraint. All modified preconditioned constraints are appended to the active list LA `(B)`.

3. The set of primitive constraints is sent to the non-linear solver and appended to NA `(C)`.

4. After all constraints in a derivation step are collected, the linear constraint solver will be invoked first.

5. Assume that we have collected r linear equality constraints of c variables. Remove a preconditioned linear constraint from the active list LA, say the k-th one of the linear system, the value of the k-th, and the $(r+1)$-st to the c-th variables are updated using equation (8). If any of the variables becomes empty, the constraint solving step fails. Otherwise, if any of the variables is changed, the constraints in both solvers that share that variable will be appended to LA and NA accordingly `(D)`.

6. Repeat step 5 until LA becomes empty.

7. In the non-linear solver, we apply interval narrowing [15] to make all the constraints there become stable. If interval narrowing fails, the constraint solving step fails. Otherwise, if a variable is further narrowed and it is also involved in some linear constraints, those constraints will be appended to LA `(E)`.

8. Repeat steps 5 to 7 in a round-robin fashion until both LA and NA become empty.

Algorithm 3. Interaction Scheme for the Two Solvers

CIAL (Alpha) [13] employs generalized interval Gaussian elimination in its linear solver. The proposed preconditioned interval Gauss-Seidel method has been incorporated into CIAL 1.x [14]. 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. All the prototypes use interval narrowing with splitting in solving inequality and non-linear constraints.

We compare our four CIAL prototypes with BNR Prolog v3.1.0 [6, 47], CLP(BNR) (or BNR Prolog v4.2.3) [45, 8], Echidna Version 0.947 beta [52, 50], ICL [33], and CLP($\mathcal{R}$) Version 1.2 [26, 30] over four numerical examples of various types: analysis of a simple DC circuit, inconsistent simultaneous equations, the ball collision problem [28], and large systems of linear equations. The examples range from purely linear constraints, to a mixture of linear and non-linear constraints, and to purely non-linear constraints. BNR Prolog runs on an Apple Mac II ($\sim$ 2 VAX MIPS with 5MB Ram) and the other systems run on a SUN SPARCstation 10 model 30 ($\sim$ 49 VAX MIPS with 32MB Ram).

7.1. Analysis of DC Circuit

Electrical engineering is an important application area for constraint logic programming [25]. Consider the simple DC circuit in Figure 4. We are interested in the currents passing through the resistors.

Assume that $V = 10$ volts and $R_i = i\ \Omega$ for $i = 1, 2, \cdots, 9$. The following system of linear equations are obtained from nodal and mesh analysis.

$$\begin{cases} \mathtt{I_s} - \mathtt{I_1} - \mathtt{I_2} - \mathtt{I_8} = 0\ , & \mathtt{I_1} = 10 \\ -\mathtt{I_s} + \mathtt{I_1} + \mathtt{I_7} = 0\ , & 2\mathtt{I_2} - 3\mathtt{I_3} - 8\mathtt{I_8} = 0 \\ \mathtt{I_2} + \mathtt{I_3} - \mathtt{I_5} = 0\ , & 3\mathtt{I_3} + 5\mathtt{I_5} - 9\mathtt{I_9} = 0 \\ -\mathtt{I_3} - \mathtt{I_4} + \mathtt{I_8} - \mathtt{I_9} = 0\ , & -4\mathtt{I_4} + 6\mathtt{I_6} + 9\mathtt{I_9} = 0 \\ \mathtt{I_4} + \mathtt{I_6} - \mathtt{I_7} = 0\ , & -\mathtt{I_1} + 4\mathtt{I_4} + 7\mathtt{I_7} + 8\mathtt{I_8} = 0 \\ \mathtt{I_5} - \mathtt{I_6} + \mathtt{I_9} = 0 & \end{cases}$$

There are 11 linear equations but only 10 unknown variables. The redundant equation cannot be located in advance, however. CIAL (Alpha) gives the following results in 1.11s.

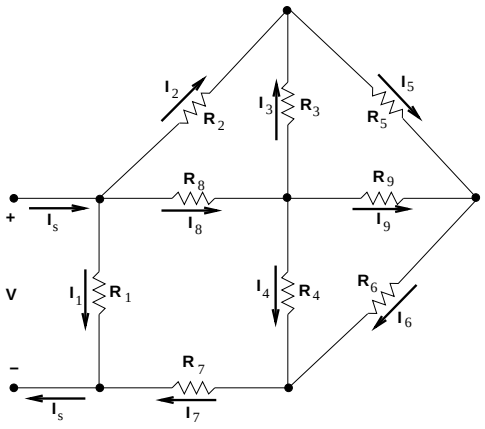

Figure 4. A Simple DC Circuit

$$\begin{array}{lll} \mathtt{I_s} \in (10.8282985772, 10.8282985773) & \mathtt{I_5} \in (0.2572597934, 0.2572597935) \\ \mathtt{I_1} \in [10.0000000000, 10.0000000000] & \mathtt{I_6} \in (0.2962385499, 0.2962385500) \\ \mathtt{I_2} \in (0.5690898460, 0.5690898461) & \mathtt{I_7} \in (0.8282985772, 0.8282985773) \\ \mathtt{I_3} \in (-0.3118300527, -0.3118300526) & \mathtt{I_8} \in (0.2592087312, 0.2592087313) \\ \mathtt{I_4} \in (0.5320600272, 0.5320600273) & \mathtt{I_9} \in (0.0389787565, 0.0389787566) \end{array}$$

CLP($\mathcal{R}$) responds in less than 1/60 second. CLP($\mathcal{R}$)'s efficiency over CIAL (Alpha) is due to the fact that generalized interval operations are time-consuming [13]. The solutions given by CLP($\mathcal{R}$) are, however, unsound.

With initial value `[-100,100]` for all variables, all the CIAL 1.x prototypes give the same results[8] as in CIAL (Alpha) and in less than 1/60 second.

By splitting 4 variables ($\mathtt{I_s}, \mathtt{I_1}, \mathtt{I_2}, \mathtt{I_3}$), ICL exits abruptly after 2 minutes of execution; Echidna (in high precision) and BNR Prolog cannot terminate in 2 and 24 hours respectively. CLP(BNR) cannot give any solution (except $\mathtt{I_1}$) with width less than 100, although all variables are specified to split. A more sophisticated use of the `precision(n)` operator [52] in Echidna is to start with a coarse precision and fine tune the precision while guiding the search simultaneously. This requires a lot of insight into the problem itself. Adopting an iterative refinement approach, anonymous referee C [49] uses Echidna to solve this problem in under 30 minutes with 4 decimal places of accuracy.

This example demonstrates the inability of interval narrowing with splitting to solve even small systems of linear constraints.

GlobSol[9] is a self-contained Fortran 90 package that finds all, rigorously verified, solutions to constrained and unconstrained global optimization problems, as well as all, rigorously verified, solutions to algebraic systems of equations. Users need to input only the objective and constraints, in a Fortran 90 program using, essentially, standard Fortran syntax. GlobSol is also able to solve the DC Circuit problem

although with 14 splits. This is not surprising since GlobSol employs a wealth of other interval and numerical techniques, such as an innovative method of finding approximate feasible points, epsilon inflation in verification, set complementation techniques to avoid large clusters of boxes, in addition to constraint propagation and the interval Newton methods. While GlobSol is a powerful tool geared for solving global optimization problems, it is based on the imperative programming paradigm. In the DC Circuit example, CIAL requires little programming because of its declarative nature. Users only have to state the 11 equalities and constraint solving is performed automatically. In GlobSol, a small amount of algorithmic programming in Fortran is still needed. The other major difference is that GlobSol operates in batch mode: all constraints must be collected in one go before execution begins. The design of CIAL caters to the incremental nature of the execution of constraint logic programs.

7.2. Inconsistent Simultaneous Equations

The following ad hoc constraint system is obviously inconsistent, since `B` and `C` should be equal and with value either 1 or -2 according to the first three constraints. Neither value is, however, consistent with the initial bound of `B`.

$$\begin{cases} \mathtt{A}+\mathtt{C} &= \mathtt{D} \\ \mathtt{A}+\mathtt{B} &= \mathtt{D} \\ \mathtt{C}(\mathtt{C}+1) &= 2 \\ \mathtt{A} \in (0,\infty), \mathtt{B} \in (-\infty,-5) \end{cases}$$

With interval splitting on all variables, CLP(BNR) returns "yes;" ICL and BNR Prolog do not terminate in 1 hour; Echidna returns "yes" with default precision and exits abruptly with high precision. CLP($\mathcal{R}$) gives "maybe" with answer constraints.

All four CIAL prototypes can detect the inconsistency without using splitting but with the cooperation of two solvers.

It is interesting to find that given the constraint `C (C + 1) = 2`, none of CLP(BNR), BNR Prolog, and Echidna can calculate the value of `C` without using splitting, even when the initial guess `[-100,100]` is given.

7.3. Collision Problem

The collision problem and the following program are adopted from [28]. This example demonstrates the non-linear constraint solving ability of CIAL. The program describes two objects, one stationary cubic wall and a ball moving along a quadratic space curve. It tries to find the time that the ball hits the wall.

```
% object_A/3 describes the shape of the wall
object_A(X,Y,Z) :-
       X <= 0, Y <= 0, Z <= 0.
% object_B/3 describes the shape of the ball
object_B(X,Y,Z) :-
```

```
        X^2 + Y^2 + Z^2 =:= 1.
% center_B/4 gives the position of the center of the ball
% at time T
center_B(T,Cx,Cy,Cz) :-
        Cx =:= T^2 - 10, Cy =:= 2*T - 10,
        Cz =:= T^2 - 7*T + 10.
% object_B_moving/4 gives the point (X,Y,Z) that is in the
% ball at time T
object_B_moving(T,X,Y,Z) :-
        center_B(T,Cx,Cy,Cz), object_B(X-Cx,Y-Cy,Z-Cz).
```

Given the following query,

```
?- T >= 0, object_A(X,Y,Z), object_B_moving(T,X,Y,Z).,
```

all CIAL prototypes give the result,

$$\mathtt{T} \in (1.6972243622, 3.3166247904).$$

It is the same as the results obtained from RISC-CLP(Real) [28], which employs symbolic algebraic method for constraint solving.

$$\begin{cases} \mathtt{T} <= 3, \\ \mathtt{T}^2 - 7\mathtt{T} + 9 <= 0. \end{cases} \quad \text{or} \quad \begin{cases} \mathtt{T} >= 3, \\ \mathtt{T}^2 - 11 <= 0. \end{cases}$$

RISC-CLP(Real) cannot solve the above quadratic equations. We use some symbolic algebra packages to solve the two systems. The union of the answers is

$$\mathtt{T} \in (1.6972243622, 3.3166247904).$$

CIAL cannot give this sharp result if we do not use the `square` [12] primitive constraint.

Both BNR Prolog, CLP(BNR), and ICL return similar results as CIAL. This is predictable since their solvers are also based on interval narrowing. Echidna returns a wide answer at low precision and exits abruptly at higher precision.

For efficiency reason, CLP($\mathcal{R}$) does not provide non-linear constraint solving. All non-linear arithmetic constraints are classified as hard constraints, which will be considered only when they become linear [40]. In this example, since no non-linear constraints can become linear, they are delayed indefinitely. The output of CLP($\mathcal{R}$) is

```
0 <= T, X <= 0, Y <= 0, Z <= 0,
-_t12 * _t12 - (Y - 2*T + 10) * (Y - 2*T + 10) + 1 = _t10 * _t10,
-_t12 + Z + 7*T - 10 = T * T, -_t10 + X + 10 = T * T.
```

7.4. Discussion

The last example on large-scale linear systems is beyond the capability of the interval narrowing based systems, i.e. ICL, BNR Prolog, CLP(BNR), and Echidna. We compare only the four CIAL prototypes and CLP($\mathcal{R}$). Before going into the last example, we give a brief discussion on the previous benchmarking results.

The CIAL solvers subsume the constraint transformation (Preconditioning or Gaussian elimination) and the interval narrowing technology. CLP($\mathcal{R}$) delays non-linear constraints from consideration and interval narrowing fails to handle even small system of linear constraints, as shown in Section 7.1. Thus, CLP($\mathcal{R}$), BNR Prolog, CLP(BNR), Echidna, and ICL are deficient in solving mixtures of linear and non-linear constraints. A simple example can be obtained by adding the constraint $\mathtt{I_x}(\mathtt{I_x} - \mathtt{1}) = \mathtt{I_s}$ to the system in Section 7.1.

7.5. Large Systems of Simultaneous Equations

We compare our four CIAL prototypes among themselves and against CLP($\mathcal{R}$) over larger scale systems and incremental execution. A 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 $[-10000, 10000]$ for all variables and calls predicate `p1/`n. Each subsequent predicate `p`i`/`n submits *one* constraint with n variables to the solver and calls `p(`$i + 1$`)/`n. That means that the `p`i`/`n predicates form a "chain" and one constraint is added to the constraint solver in each derivation step. This call pattern exercises the incrementality of the linear solver to the fullest extent. For example, the program for rank 50 looks like the following:

```
t :-
   X0>= -10000,X0<=10000,
   X1>= -10000,X1<=10000,
     ⋮    ⋮
   X49>= -10000,X49<=10000,
   1 randomly generated constraint with 50 variables,
   p1(X0,X1,X2,...,X49).
p1(X0,X1,X2,...,X49) :-
   1 randomly generated constraint with 50 variables,
   p2(X0,X1,X2,...,X49).
  ⋮      ⋮      ⋮
p48(X0,X1,X2,...,X49) :-
   1 randomly generated constraint with 50 variables,
   p49(X0,X1,X2,...,X49).
p49(X0,X1,X2,...,X49) :-
   1 randomly generated constraint with 50 variables.
```

Table 1. Computation Time and Speedup

Rank	CIAL 1.0 (Beta)	CIAL 1.1	Speedup	CIAL 1.1a	CLP($\mathcal{R}$)
10	0.35s	0.35s	1.00	0.19s	0.01s
20	2.87s	2.20s	1.30	1.69s	0.08s
30	12.99s	8.28s	1.57	4.07s	0.35s
40	38.50s	24.33s	1.58	11.22s	0.93s
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 1 shows the computation time for problems of size ranging from 10 to 100. As expected, CIAL 1.1 exhibits good 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 decimal places 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 1.x's. At the time of implementation and because of our choice of implementation language (for compliance with the CLP($\mathcal{R}$) implementation), we could not adopt the best possible interval arithmetic implementations, such as the one available in the Sun Forte Fortran 95 compiler[10]. All our CIAL implementations are based on a home-grown rudimentary library of interval arithmetic operations. 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 return answer intervals of useful width on linear systems of rank greater than 30.

8. Concluding Remarks

8.1. Summary and Contributions

In this paper, we have discussed the deficiencies of interval narrowing with splitting. Our experiments show that interval narrowing based systems fail to solve even small problems efficiently and effectively. Thus interval narrowing with splitting is im-

practical in solving general interval constraints over the real domain. We propose to separate linear equality constraint solving from inequality and non-linear constraint solving. We have extended and generalized the traditional preconditioned interval Gauss-Seidel method, resulting in the incremental preconditioned interval Gauss-Seidel method. This technique has been implemented in the linear constraint solver of our new interval constraint logic programming system, CIAL (for *Constraint Interval Arithmetic Language*), which shares the same declarative and operational semantics as those of ICLP($\mathcal{R}$) [34]. We have designed an architecture for CIAL and established the interaction among the modules in the architecture. Unification between interval variables and other terms is handled in an extended unification algorithm. Input arithmetic constraints are decomposed into linear equalities and a set of convex primitive constraints. The former is handled by the linear solver, while we apply interval narrowing on the latter in the non-linear solver. We have constructed several CIAL prototypes and compared them with several major interval constraint logic programming languages.

The contribution of our work is three-fold. First, we have derived an efficient interval linear equality solver for incremental execution in interval CLP languages. Its correctness has been established. Also, the solving method has worst-case time complexity of $O(n^2(m+n))$ complexity where m and n are the number of constraints and variables respectively. Our complexity is the same as that of the incremental Gaussian elimination used in CLP($\mathcal{R}$). Solutions given by the incremental preconditioned interval Gauss-Seidel method are, however, slightly wider than those obtained from the non-incremental method. Our large scale experiments show that solutions given by this incremental method still reach 8 decimal places of accuracy in general.

Second, we have shown how an interval linear solver can be incorporated into a system which has already had a non-linear solver based on interval narrowing. We have derived a constraint decomposition procedure and an interaction scheme for the two solvers. Input constraints are divided into two categories, which will be sent to two solvers accordingly. The two solvers share common variables, interact in a round-robin fashion, and cooperate towards solving a system of numerical constraints. We have shown the termination of the interaction scheme.

Third, we have constructed several prototypes of CIAL and compared them with one another, as well as with several existing interval constraint logic programming languages. Of the four prototypes, CIAL 1.1 (Beta) and CIAL 1.1a have been shown to be the most efficient one in solving large scale linear systems. In the comparisons of CIAL and other existing systems, CIAL is competitive in various aspects: all prototypes are substantially more efficient and can solve more classes of problems than any other existing systems when used alone.

8.2. Future Work

A number of questions remain to be investigated. The linear solver can only handle linear equalities. It would be interesting to investigate how linear inequalities can be accommodated.

On the theoretical side, it would be interesting to study the level of interval consistency attainable in the incremental preconditioned interval Gauss-Seidel method. It should reach a consistency level falling between box consistency and hull consistency [7].

Concerning implementations, our CIAL prototypes have much to be desired. First, the CIAL architecture is 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 separate from the Prolog engine. Communications between the solvers and the Prolog engine incur high overhead. Backtracking also becomes a costly operation. We expect that the work of Lee and Lee [33] can be used as basis to integrate the interval constraint solving and the Prolog engine at the Warren Abstract Machine (WAM) level. Third, the built-in constraints in CIAL are limited. To apply CIAL on real-life problems (e.g. scheduling), more relations, such as `max/2`, `min/2`, `sin/1` [45], should be provided.

To establish the practicality of our approach, we need to try CIAL on more real-life applications, e.g. job shop scheduling, process planning, assembly line balancing, temporal and spatial reasoning, multiagent planning, finite element modeling, circuit design, etc.

Acknowledgments

We thank Spiro Michaylov, Roland Yap, and Tak-wai Lee for numerous discussions and critical comments. We are indebted to insightful comments by anonymous referees of ILPS94, ICLP95, the Journal of Logic Programming, and the Reliable Computing Journal. R. Baker Kearfott has been most helpful and patient in dealing with our submission to Reliable Computing. 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.

Notes

1. Prolog III provides the option of using floating-point arithmetic, although the default is rational arithmetic.
2. If p is an n-ary predicate symbol (a function returning true or false) and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atom* or *atomic formula* [37]. While terms denote objects, atoms denote relations among terms.
3. A *rule* in an interval constraint logic program has the form H `:-` $\vec{\Theta'}, \vec{\Delta'}$, where H is an atom, $\vec{\Theta'}$ is a set of constraints, $\vec{\Delta'}$ is a set of atoms, and `:-` is a symbol to mean "if." The rule reads if $\vec{\Theta'}$ and $\vec{\Delta'}$ are true, then H is also true.
4. We do not consider such special divide-and-conquer *square* matrix multiplication algorithms as Strassen's algorithm ($O(n^{2.81})$) [17]. 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.

5. 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.
6. $A^I \otimes (B^I \oplus C^I) \subseteq A^I \otimes B^I \oplus A^I \otimes C^I$.
7. $(A^I \otimes B^I) \otimes C^I = A^I \otimes (B^I \otimes C^I)$.
8. CIAL 1.1a always gives closed bounds in its answers.
9. `http://www.mscs.mu.edu/~globsol/`
10. `http://www.sun.com/forte/fortran/interval/`

References

1. A. Aggoun and N. Beldiceanu. Overview of the CHIP compiler system. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 775–789, Paris, France, 1991.
2. A. Aiba, K. Sakai, Y. Sato, D.J. Hawley, and R. Hasegawa. Constraint logic programming language CAL. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 263–276, Tokyo, Japan, 1988.
3. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, 1991.
4. G. Alefeld and J. Herzberger. *Introduction to Interval Computations*. Academic Press, 1983.
5. 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*. Society for Industrial & Applied Mathematics, 1993.
6. Bell-Northern Research Ltd. *BNR Prolog Reference Manual*, 1988.
7. F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Logic Programming: Proceedings of the 1994 International Symposium*, pages 124–138, Ithaca, USA, 1994.
8. F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32:1–24, 1997.
9. C. Bessiere. Arc-consistency and arc-consistency again. *AI Journal*, 65(1):179–190, 1994.
10. B. Buchberger. Gröbner bases: An algorithmic method in polynomial ideal theory. In N.K. Bose, editor, *Recent Trends in Multidimensional Systems Theory*, chapter 6. D. Riedel Publ. Comp., 1983.
11. B. Buchberger and H. Hong. Speeding-up quantifier elimination by Gröbner bases. Technical Report 91-06.0, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz, Austria, 1991.
12. 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.
13. 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*, pages 109–123, Ithaca, USA, 1994.
14. C.K. Chiu and J.H.M. Lee. Interval linear constraint solving using the preconditioned interval Gauss-Seidel method. In *Proceedings of the 12th International Conference on Logic Programming*, pages 17–31, Kanagawa, Japan, 1995.
15. J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.
16. A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, July 1990.
17. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, eighth edition, 1992.
18. E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.
19. 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*, pages 693–702, Tokyo, Japan, 1988.

20. D.M. Gay. Solving interval linear equations. *SIAM Journal on Numerical Analysis*, 19(4):858–870, 1982.
21. E. Hansen. *Global Optimization using Interval Analysis*. Marcel Dekker, Inc., 1992.
22. E.R. Hansen. Interval arithmetic in matrix computations. *SIAM Journal on Numerical Analysis*, 2:308–320, 1965.
23. E.R. Hansen. Bounding the solution of interval linear equations. *SIAM Journal on Numerical Analysis*, 29(5):1493–1503, October 1992.
24. W.S. Havens, S. Sidebottom, J. Jones, M. Cuperman, and R. Davison. Echidna constraint reasoning system: Next-generation expert system technology. Technical Report CSS-IS TR 90-09, Centre for Systems Science, Simon Fraser University, Burnaby, B.C., Canada, 1990.
25. N. Heintze, S. Michaylov, and P. Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. *Journal of Automated Reasoning*, 9(2):231–260, October 1992.
26. N.C. Heintze, J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. *The CLP($\mathcal{R}$) Programmer's Manual Version 1.2*. IBM Thomas J Watson Research Center, 1992.
27. H. Hong. *Improvements in CAD-Based Quantifier Elimination*. PhD thesis, The Ohio State University, 1990.
28. H. Hong. Non-linear real constraints in constraint logic programming. In *Proceedings of the Second International Conference on Algebraic and Logic Programming*, pages 201–212, 1992.
29. J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM POPL Conference*, pages 111–119, Munich, January 1987.
30. J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP($\mathcal{R}$) language and system. In *ACM Transactions on Programming Languages and Systems*, volume 14, pages 339–395, 1992.
31. R.B. Kearfott. *Rigorous Global Search, Continuous Problems*. Kluwer Academic Publishers, 1996.
32. J.H.M. Lee. *Numerical Computation As Deduction In Constraint Logic Programming*. PhD thesis, Department of Computer Science, Logic Programming Laboratory, University of Victoria, Victoria, Canada, 1992.
33. 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*, pages 122–128, New Orleans, USA, 1994.
34. J.H.M. Lee and M.H. van Emden. Adapting CLP($\mathcal{R}$) to floating-point arithmetic. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, volume 16, pages 996–1003, Tokyo, Japan, 1992.
35. J.H.M. Lee and M.H. van Emden. Interval computation as deduction in CHIP. *Journal of Logic Programming*, 16:255–276, 1993.
36. O. Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993.
37. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.
38. A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.
39. A.K. Mackworth, J.A. Mulder, and W.S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.
40. 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.
41. R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.
42. R.E. Moore. *Methods and Applications of Interval Analysis*. SIAM, 1979.
43. A. Neumaier. Overestimation in linear interval equations. *SIAM Journal on Numerical Analysis*, 24(1):207–214, February 1987.
44. A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.
45. W. Older. Constraints in BNR Prolog. Technical Report Draft 01, Software Engineering Centre, Bell-Northern Research, Ottawa, Canada, 1993.
46. 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.

47. W. Older and A. Vellino. Constraint arithmetic on real intervals. In A. Colmerauer and F. Benhamou, editors, *Constraint Logic Programming: Selected Research*, pages 175–196. MIT Press, 1992.
48. W.J. Older. The application of relational arithmetic to X-ray diffraction crystallography. Technical Report 89001, Software Engineering Centre, Bell-Northern Research, Ottawa, Canada, 1989.
49. Journal of Logic Programming Referee C. Personal communication, November 1995.
50. G. Sidebottom and W.S. Havens. Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 8(4):601–623, 1992.
51. G.A. Sidebottom. *A Language for Optimizing Constraint Propagation*. PhD thesis, School of Computer Science, Simon Fraser University, November 1993.
52. 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.
53. L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, second edition, 1994.
54. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, London, England, 1989.
55. P. Van Hentenryck. A general introduction to NUMERICA. *Artificial Intelligence*, 103(1–2):209–235, 1998.
56. P. Van Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2), 1997.
57. P. Van Hentenryck, L. Michel, and D. Kapur. *Numerica: a Modeling Language for Global Optimization*. The MIT Press, 1997.
58. P. Vasey. Qualified answers and their application to transformation. In *Proceedings of the Third International Logic Programming Conference*, pages 425–432, 1986.