A WAM-based Abstract Machine for Interval Constraint Logic Programming

J.H.M. Lee and T.W. Lee

Department of Computer Science
The Chinese University of Hong Kong
Hong Kong, Shatin
{jlee,twlee}@cs.cuhk.hk

Abstract

In this paper, we propose an integration of constraint interval arithmetic into logic programming at the machine architectural level, of which WAM is the de facto standard. The language in consideration is ICL, a subset of ICHIP which shares the same semantic properties as CHIP. We use the work of Diaz and Codognet as a starting point, and exploit the simplicity of interval constraint solving over finite domain constraint solving. The resulting extension of WAM is simple, efficient, robust, and portable. Our ICL prototype compares favourably against BNR Prolog, CLP(BNR), Echidna, and CLP($\mathcal{R}$) in various types of numerical examples.

1 Introduction

Arithmetic facilities in Prolog, which are unsound and functional in nature, have much to be desired. Cleary [5] realizes the shortcomings and proposes a relational version of interval arithmetic for Prolog. Loosely based on Cleary's ideas, several constraint logic programming languages [15, 3, 17, 13, 11, 12], such as BNR Prolog and Echidna, have emerged and have demonstrated their applicabilities in areas such as critical path scheduling [15], X-ray diffraction crystallography [15], boolean constraint solving [3], disjunctive constraint solving [3, 17], and finite domain constraint solving [3].

The implementation of these languages either employs meta-programming techniques [16, 13, 11] or consists of an independent constraint solver built as a module separate from the Prolog engine [17, 3]. Meta-programming techniques consist of building a constraint solver as a module on top of Prolog. This approach is good for fast prototyping but lacks the efficiency required for real applications. The independent solver approach provides a clean interface between the Prolog kernel and the constraint solver. Communications between the solver and the Prolog engine, however, incur high overhead. Backtracking also becomes a costly operation.

Attaining an efficient implementation calls for a tight integration of interval constraint solving and the Prolog engine at the machine architectural level. **W**arren's **A**bstract **M**achine (WAM for short) [21], which has been a de facto standard of Prolog implementation, is chosen as our base architecture. Different approaches are proposed to extend the WAM architecture to handle constraints efficiently for such languages as CLP($\mathcal{R}$) [10], CHIP [2], and `clp(FD)` [6]. The design of these abstract machines is strongly influenced by the computation domains and the constraint solving techniques adopted.

We consider the **I**nterval **C**onstraint **L**anguage (ICL for short), a subset of ICHIP [12]. ICL is based on constraint interval arithmetic [11, 12], a generalization of Cleary's framework which consists of an interval narrowing operation and a relaxation algorithm. Lee and van Emden [12] show that the interval narrowing operation is a form of LAIR [19], and that the relaxation algorithm is equivalent to the Lookahead Efficient Computation rule [19]. This semantic resemblance suggests using either the work of Aggoun and Beldiceanu [2] or Diaz and Codognet [6] as a starting point in our project. We adopt the latter. Our design exploits the simplicity of the interval domain structure, the need to handle only LAIR (interval narrowing), and the embedding of the computation rule in the relaxation algorithm.

The rest of the paper is organized as follows. The syntax and semantics of the ICL language are presented in section 2. In section 3, we describe the extension to the WAM architecture and the enhanced instruction set. In section 4, we discuss the ICL prototype implementation and benchmarking results. We summarize our contributions and discuss future perspectives in section 5.

2 The ICL language

ICHIP [12] extends CHIP [7] with constraint interval arithmetic by allowing domain [19] to be, besides finite set of values, interval of real numbers. ICHIP provides relational arithmetic on integers (by finite domain), rational and real numbers (by interval domain). We restrict our attention to ICL, the real number subset of ICHIP.

2.1 Syntax and declarative semantics

An *interval* is a segment, possible infinite, of the real number line. Let I be an interval. X^I is an *interval domain variable*, abbreviated as *i-variable*, with interval domain I. Intuitively the variable X^I can be bound only to the elements of I. For example, `X^'[-0.123,4.56)'` is an i-variable with interval domain $\{x \mid -0.123 \leq x < 4.56\}$. We call the usual variables of first-order logic *Herbrand variables*, abbreviated as *h-variables*.

ICL provides a set of primitive constraints on real numbers, namely `add/3`, `multplus/3`, `multminus/3`, `lt/2`, `le/2`, `sqrplus/2` and `sqrminus/2` as described in [12]. Other constraints that satisfy the *interval integrity*[1] criterion [11, 12] can be added to this set as appropriate. The predicate names of the constraints are self-explanatory.

An *atom* is any standard atomic formula whose predicate symbol is different from those of the primitive constraints, and can contain both i-variables and h-variables. A *clause* in ICL has the form

$$H \texttt{ :- } C_1, \ldots, C_m, B_1, \ldots, B_n. \qquad (m, n \geq 0)$$

where H is an atom, $C_1, \ldots, C_m$ is a set of primitive constraints, and $B_1, \ldots, B_n$ is a set of atoms[2]. H is called the *head* and $C_1, \ldots, C_m, B_1, \ldots, B_n$ is the *body* of the clause. The specification of the domain of an i-variable should appear at most once. A *query* is a clause without the head.

The declarative semantics of ICL resembles that of CHIP. Variables are universally quantified. The "`:-`" symbol is read as *if*, and the commas are read as conjunction. If a clause has no body, the "`:-`" can be omitted. A *program* in ICL is a set of clauses. We conclude this subsection by an ICL program for standard mortgage calculation in figure 1. The program is almost identical to the CLP($\mathcal{R}$) version, except that constraints are defined using primitive constraints[3]. The ">=" predicate is for the comparison of h-variables and integers. Output from the query "`?- mg(99999,10,I^'[0.01,0.05]',B,5000).`" is:

```
I ∈ (0.0099999999,0.0500000001)
B ∈ (58150.0452133925,99998.3711053734)
```

```
mg(P, 1, I, B, MP) :-
    multplus(I, P, T1), add(P, T1, T2),
    add(B, MP, T2).

mg(P, T, I, B, MP) :-
    T >= 2, add(1, I, T1),
    multplus(T1, P, T2), add(TA, MP, T2),
    add(TB, 1, T), mg(TA, TB, I, B, MP).
```

Figure 1: An ICL program for mortgage calculation.

2.2 Computation model

The introduction of i-variables calls for an enhanced unification algorithm. The extension is similar to that of CHIP [19]. We have three cases to consider. (1) Unification for two i-variables X^I and Y^J succeeds with binding $\{X^I/Z^{I\cap J}, Y^J/Z^{I\cap J}\}$ if $I \cap J \neq \emptyset$. Otherwise, unification fails. (2) If a h-variable X is unified with an i-variable Y^J, unification succeeds with binding $\{X/Y^J\}$. (3) If an i-variable X^I is unified with a constant c, unification succeeds with binding $\{X^I/c\}$ if $c \in I$. Otherwise, unification fails.

We are now ready to define the *ICL Computation Model*. Let P be an ICL program and Q_i a query. Q_{i+1} is *derived* from $Q_i = \leftarrow \vec{C}, B_1, \ldots, B_j, \ldots, B_q$, where $\vec{C}$ is a conjunction (possibly empty) of primitive constraints, and $B_1, \ldots, B_q$ is a conjunction of atoms, if the following conditions hold:

1. B_j is the *selected* atom in Q_i;

[1] No name was given to this criterion in [11, 12]. An explanation of the criterion requires a full exposition of interval narrowing, which is outside the scope of this paper. For example, the `multiply/3` relation does not satisfy the criterion since interval narrowing using `multiply/3` may generate interval domains that are unions of disjoint intervals.

[2] The order of the primitive constraints and body atoms is not important. They can be mixed together.

[3] This syntactic deficiency can be eliminated by a simple front-end preprocessor that translates mathematical constraints into primitive constraints, possibly introducing a few slack variables.

2. there exists a clause "H:-$\vec{C'}, A_1, \ldots, A_n$." ($n \geq 0$) in P such that H is unifiable with B_j generating substitution θ_i;

3. the relaxation algorithm [12] stabilizes for constraint network $\vec{C''} = (\vec{C} \cup \vec{C'})\theta_i$;

4. $Q_{i+1} = \leftarrow \vec{C''}, (B_1, \ldots, B_{j-1}, A_1, \ldots, A_n, B_{j+1}, \ldots, B_q)\theta_i$.

A *derivation* is a, possibly infinite, sequence of queries $Q = Q_0, Q_1, Q_2, \ldots$ such that Q_{i+1} is derived from Q_i generating substitution θ_i. A derivation is *successful* if it is finite with m steps and the last query contains only a stable constraint network N. We call N the *answer network*. A *computed answer* of a successful derivation is the composition of substitutions $\theta = \theta_1 \cdots \theta_m$ plus the *domain constraints* of the form $X \in I$ for each i-variable X^I in N.

2.3 Answer interpretation and splitting

The semantics of CHIP, and thus ICL, is based on arc consistency techniques [14]. It is well-known that arc consistency techniques are "incomplete" [14]: a network can be stable but neither a solution nor inconsistency is found. Logically, we interpret constraints in the answer network as qualified or conditional answers [20]. Let P be a logic program with domain variables and $\leftarrow G$ a goal. Suppose, we have derived from $\leftarrow G$ a non-empty goal $\leftarrow G_1, \ldots, G_n$, with θ being the composition of all substitutions so far. The clause $(G \leftarrow G_1, \ldots, G_n)\theta$ is a *conditional answer* to the original goal. From the soundness of SLDD-resolution and LAIR [19], we have

$$P \models \forall(G \leftarrow G_1, \ldots, G_n)\theta$$

where the universal quantification is over all variables that occur in $(G \leftarrow G_1, \ldots, G_n)\theta$. This interpretation is also adopted for answer constraints in CLP languages [9].

Sometimes, constraint interval arithmetic is too weak to narrow intervals to our desirable width. For example, this behaviour occurs if we use ICL to solve the roots of general polynomials and simultaneous equations [5]. ICL provides two built-in predicates `split/1` and `split/2` to narrow interval domain to the smallest width allowed by the (preset) precision of the underlying floating-point system. The two predicates are based on *domain splitting*, a case analysis technique. They are only invoked at the end of a derivation, and work by splitting of the domain of the argument variable into two partitions, visit one of them, and the other upon backtracking. The idea is that splitting destabilizes the answer network and the relaxation algorithm will be re-invoked. This splitting process is repeated until the interval domain in consideration is narrowed to the desirable width.

3 The abstract machine

In this section, we describe the integration of constraint interval arithmetic into WAM and present the compilation scheme. Our machine architecture is similar to that of CHIP and `clp(FD)`. We deviate from Diaz and Codognet [6] in the way that we encode constraints and handle the multiple-trailing problem. First, the primitive constraints used by Diaz and Codognet are low level. Compositions of these constraints can be used to model more complex constraints than necessary for constraint interval arithmetic. Our primitive constraints are direct encoding of interval constraint solving. Second, Diaz and Codognet use the time stamp method to handle multiple-trailing, which incurs extra register and extra heap space consumption. Our approach involves only a simple address comparison.

3.1 Modifying WAM for i-variables

The new data type i-variables is stored in the heap with an appropriate tag. Associated with an i-variable is an i-variable frame which will be described in section 3.2. The introduction of i-variables prompts for modifications of data manipulation, unification, indexing and trailing.

Data manipulation As stated in [6], finite domain variables in `clp(FD)` cannot be duplicated (as is done for terms by structure-copying). When a source word W_s must be loaded in a destination word W_d, if W_s is a finite domain variable then W_d is bound to W_s else W_s is physically copied in W_d. The same scheme is applicable to i-variables.

Unification Unification is enhanced as described in section 2.2. The modification is similar to that of `clp(FD)` [6]. The change also slightly affects instructions which manipulate constant such as `set_constant`.

Indexing For procedure definitions that contain more than one clause, WAM adopts indexing to optimize clause selection. The first argument of a clause's head is used as the indexing key. Thus, if a goal's first argument is an i-variable, try all clauses whose key is a variable (i-variable or h-variable) or a number.

Trailing A simple-entry trail is used in WAM as unbound variables can be trailed at most once and the value of an unbound variable is defined by its address due to self-references. For i-variables, the previous values need to be stored in the trail. Thus, a tagged trail is needed to distinguish the two types of objects in the trail: one-word entry for h-variables, and two-word entry for trailing one previous value. In `clp(FD)`, there is one more type of object in the trail: $(n+2)$-word entry for trailing n previous values [6]. As the interval of an i-variable is gradually shrunk, the WAM criterion for trailing introduces the multiple-trailing problem. `Clp(FD)` uses a time stamp method [1] to solve the problem. We co-develop and adopt the trailing scheme in [18].

3.2 Data structures for constraints

In `clp(FD)`, an argument frame is created to represent the environment in which the constraint is called. It is used to achieve optimization 1 in [6]. In ICL, there is no similar data structures. On the other hand, ICL has its own active list (to be described below).

Constraint frame A constraint frame is created for every constraint and consists of (1) the addresses of the i-variables which are constrained, (2) the numeric constants in the constraint, and (3) the pointer to the next entry in the active list. The contents of a constraint frame differ from that of `clp(FD)` due to differences in the solvers used.

I-variable frame The frame associated with the i-variable `X` consists of (1) the address to an interval (a non-empty segment of real line), and (2) the constraints depending on `X` (a list structure). These two parts are modified, and thus trailed, independently. The interval is updated during execution whereas the list is created or updated when constraints depending on `X` are *installed* and unification takes place. The contents of an i-variable frame is simpler than that of a finite domain variable frame in `clp(FD)` as interval is a simpler data type than finite domain.

Active list The active list is a list structure containing constraints that are possibly unstable. The first heap cell is reserved so that it is always set to contain address of the first constraint frame in the active list. A new global register `TAIL` is used to contain the address of the last constraint frame in the active list. Choice points are created only when a standard Prolog goal is selected, i.e. the active list is empty. Thus, the active list needs no trailing.

3.3 Modifying WAM for interval splitting

Interval splitting is delayed until the derivation is successful. A choice point is created when an interval is split into two partitions. ICL visits one of them, and the other upon backtracking.

Split frame The split frame associated with the splitting i-variable `X` consists of (1) the address of the splitting i-variable `X` and (2) the desirable width of the interval of `X`.

Split list The split list is a list structure containing all split frames. It is created or updated in `split/1` and `split/2`.

3.4 Compilation scheme

The compilation of a clause in ICL gives rise to three kinds of new instructions:

- Creating and loading an i-variable: the address of an i-variable is loaded to a register.
  - `put_i_var interval,Vn,Ai`
    Create on the top of the heap a new i-variable with interval domain `interval` and copys its address to both register V_n and register A_i
  - `put_i_val Vn,Ai`
    If the dereferenced value of register V_n is
    - an unbound variable: Similar to `put_i_var (nInf,pInf),Vn,Ai`
    - a numeric constant: Bind register A_i to the constant
    - an i-variable: Copy its address to register A_i
  - `get_i_var interval,Vn,Ai`
    Similar to `put_i_val Ai,Vn`
- Installing a constraint: the instruction creates and installs a constraint frame in the active list. It also initializes or updates the appropriate lists for all i-variables constrained.

- `install_constr p,n`
  Create on the top of the heap a constraint frame with constraint type p and parameters stored in register A_1 to A_n. Update also the appropriate lists and the active list.

- Invoking the relaxation algorithm: the instruction invokes the relaxation algorithm.
  - `relax`
    Invokes the relaxation algorithm to reduce the current constraint network. If the relaxation algorithm fails, invoke backtracking.

Figure 2 compares the code generated for a predicate `p(X,X,Y)` when it is treated as a constraint and when it is treated as a Prolog goal.

As a Prolog goal	As a constraint
`put_variable X4, A1`	`put_i_var (nInf,pInf), X4, A1`
`put_value X4, A2`	`put_i_val X4, A2`
`put_variable X5, A3`	`put_i_var (nInf,pInf), X5, A3`
`call p/3`	`install_constr p 3`

Figure 2: Code generated for `p(X,X,Y)`

Figure 3 shows an example of code generated for a fact containing i-variables.

```
p/2 :   get_structure s/1, A1
        unify_variable X4
        get_i_var (1,2], X3, X4
        get_i_var (2,3), X5, A2
        relax
        proceed
```

Figure 3: Fragment of code generated for the fact `p(s(X^'(1,2]'),Y^'(2,3)')`

In compiling a clause, the body is first ordered so that constraints appear before any standard Prolog goals, i.e. in the form $p{:}-c_1, \ldots, c_n, q_1, \ldots, q_m$. The code generated follows the pattern shown in figure 4. Note that the instruction `relax` is also generated for a clause containing no constraint since unification can make a stable constraint unstable.

code for p
code for c_1
⋮
code for c_n
`relax`
code for q_1
⋮
code for q_m

Figure 4: Pattern of code generated for a clause

4 Prototype and benchmarking

To demonstrate the feasibility of our proposal, we have built a prototype of ICL. The compiler is written in Quintus Prolog Release 3.1 whereas the emulator is written in C. The current implementation consists of about 1600 lines of Prolog and about 3500 lines of C.

In the following, we compare ICL with BNR Prolog v3.1.0, CLP(BNR) (or BNR Prolog v4.2.3), Echidna Version 0.97, and CLP($\mathcal{R}$) Version 1.2 over three numerical examples. 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).

4.1 Simple linear simultaneous equations

Constraint interval arithmetic alone is sometimes inadequate for solving linear systems. Consider the following simple system of linear simultaneous equations in two unknowns.

$$\left\{ \begin{array}{rcl} X+Y & = & 5 \\ X-Y & = & 6 \\ X,Y & \in & (-\infty,+\infty) \end{array} \right.$$

All of BNR Prolog, CLP(BNR), Echidna and ICL cannot narrow any of the variables when no initial bound is given. When we give them an initial guess, say $X,Y \in [-1000,1000]$, they return

$$X \in [-989,1000], Y \in [-995,994]$$

When interval splitting is employed, ICL returns

$$X \in [5.5,5.5], Y \in [-0.5,-0.5]$$

which is the same as the exact solution returned by CLP($\mathcal{R}$). The other three systems return slightly less accurate solutions when interval splitting is employed.

4.2 Inconsistent simultaneous equations

The aim of this example is to show the weakness of constraint interval arithmetic, even when splitting is employed. The following is an ad hoc linear system which is inconsistent.

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

BNR Prolog and ICL do not terminate in 90 minutes. CLP(BNR) returns "yes" while Echidna returns "yes" with default precision and exits abruptly with high precision. CLP($\mathcal{R}$) can detect the inconsistency.

4.3 Wilkinson polynomial

The Wilkinson polynomial equation [8] is stated as follows:

$$\prod_{i=1}^{20}(X+i)+EX^{19}=0 \qquad (1)$$

The problem is to find the real roots of equation (1).

Consider the case where $X \in [-20,-10]$. When $E=0$, ICL computes the roots which are -20, $-19,\ldots,-11,-10$. The roots can also be computed using simple numerical analysis technique. When $E\neq 0$, the roots are perturbed.

Even if E is as small as 2^{-23}, the perturbation is already so big that all roots in $X\in[-20,-10]$ disappear. This effect is demonstrated in figure 5, which consists of the curves $Y=\prod_{i=1}^{20}(X+i)$ and $Y=-EX^{19}$. ICL returns no answer as expected. CLP(BNR) and BNR Prolog produce the same response. Echidna does not terminate in 15 minutes.

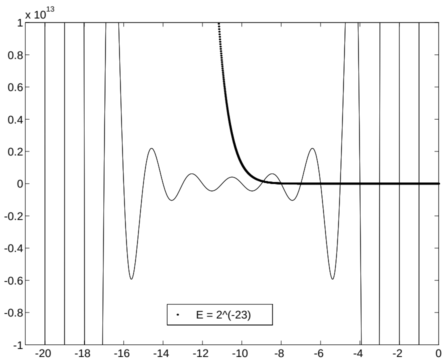

Figure 5: $Y=\prod_{i=1}^{20}(X+i)$ and $Y=-EX^{19}$

When $E=2^{-40}$, the perturbation becomes insignificant. ICL produces the first solution with more than 15 decimal places in 0.65 second. ICL gives no answer around -20 since the root there is indeed less than -20. CLP(BNR) returns similar results in 1.46 seconds. BNR Prolog solves the problem in 1.95 seconds but with only 3 decimal places of accuracy. Echidna aborts abnormally for default precision and does not terminate in 15 minutes for higher precision. CLP($\mathcal{R}$) cannot handle non-linear constraints in both cases.

4.4 Summary

There are three observations. First, CLP($\mathcal{R}$) is the most efficient for linear constraints but it incurs round-off errors. CLP($\mathcal{R}$) fails to tackle any of the non-linear systems. Second, constraint interval arithmetic excels in non-linear constraint systems but are weak with linear systems in general. Interval splitting can help in some, but not all, cases. Third, ICL is the most efficient, robust and accurate among the interval constraint logic programming languages.

5 Concluding remarks

As far as we know, this paper is the first exposition of a WAM-based abstract machine for interval constraint logic programming languages. The contribution of this paper is two-fold. First, we describe an abstract instruction set for the ICL language. Implementation results show that the set can be smoothly integrated into WAM. Second, a prototype implementation of ICL is built and benchmarking results show its numerical processing ability.

There are a few directions for future work. First, the benchmarking results show that interval narrowing is inefficient in solving linear constraints. This suggests the integration of an efficient linear interval constraint solver [4]. Second, an ICL derivation can end with floundered constraints. Consistency checking techniques are topics of further studies. Third, implementation results show that `clp(FD)` is more than twice as fast as the commercial version of CHIP on average [6]. It is interesting to check if the use of primitive constraint `X in r` can improve the performance of ICL.

Acknowledgement

We are indebted to Chong Kan Chiu, Maarten van Emden, and Bo Ming Tong for numerous discussions and critical comments. We thank the anonymous referees for their insightful comments. Access to CLP($\mathcal{R}$), BNR Prolog, CLP(BNR), and Echidna is gratefully acknowledged. This research was supported in part by a Direct Grant (A/C 220500500) and a RGC Earmarked Grant (A/C 221500260), both by the Hong Kong University and Polytechnic Grants Committee.

References

[1] A. Aggoun and N. Beldiceanu. Time stamps techniques for the trailed data in CLP systems. In *Actes du Séminaire 1990 - Programmation en Logique*, 1990.

[2] A. Aggoun and N. Beldiceanu. Overview of the CHIP compiler system. In *Proceedings of the Eight International Conference on Logic Programming*, 1991.

[3] F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer and boolean constraints. *(to appear) Journal of Logic Programming*, 1994.

[4] 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*, 1994.

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

[6] D. Diaz and P. Codognet. A minimal extension of the WAM for `clp(FD)`. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 774–790, 1993.

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

[8] 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.

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

[10] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. An abstract machine for CLP($\mathcal{R}$). In *Proceedings of the ACM SIGPLAN U92 Conference on Programming Language Design and Implementation*, pages 128–139, San Francisco, 1992. ACM Press.

[11] 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*, pages 996–1003, Tokyo, Japan, June 1992.

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

[13] O. Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, 1993.

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

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

[16] W.J. Older. Interval arithmetic specification. Research Report 89032, Computing Research Laboratory, Bell-Northern Research, Ottawa, Ont., Canada, July 1989.

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

[18] B.M. Tong and H.F. Leung. Implementation of a data-parallel concurrent constraint programming system. In *Proceedings of the First International Symposium on Parallel Symbolic Computation (to appear)*, 1994.

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

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

[21] D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, 1983.