# Interval Computation as Deduction in CHIP

J.H.M. Lee*  M.H. van Emden†

**Abstract**

Logic programming realizes the ideal of "computation is deduction," but not when floating-point numbers are involved. In that respect logic programming languages are as careless as conventional computation: they ignore the fact that floating-point operations are only approximate and that it is not easy to tell how good the approximation is. It is our aim to extend the benefits of logic programming to computation involving floating-point arithmetic.

Our starting points are the ideas of Cleary and the CHIP programming language. Cleary proposed a relational form of interval arithmetic which was incorporated in BNR Prolog in such a way that variables already bound can be bound again. In this way the usual logical interpretation of computation no longer holds. In this paper we develop a technique for narrowing intervals that we relate both to Cleary's work and to the constraint-satisfaction techniques of artificial intelligence. We then modify CHIP by allowing domains to be intervals of real numbers. To reduce arithmetic primitives with interval domains, we use our interval narrowing technique as an implementation of the Looking-Ahead Inference Rule. We show that the result is a system where answers are logical consequences of a declarative logic program, even when floating-point computations have been used. We believe ours is the first system with this property.

## 1 Introduction

The state of current programming method, as exemplified by structured programming and by languages such as C and Ada, leaves much to be desired. Software engineering can increase our confidence in software but, in large-scale and safety-critical programs, what we

*Department of Computer Science, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong. Email: jlee@garden.cs.cuhk.hk.

†Department of Computer Science, University of Victoria, Victoria, B.C., Canada V8W 3P6. Email: vanemden@csr.uvic.ca.

need is *certainty* of correctness. Towards this aim, revolutionaries such as Patrick Hayes and Robert Kowalski advocated radical change, as embodied in Hayes's motto: "Computation is deduction." [14] The declarative programming paradigm satisfies Hayes's motto. According to this approach, programs are definitions in a declarative language and every computation step is a valid inference, so that results are logical consequences of program and data. Logic programming, as realized by pure Prolog and the CLP scheme, is an example of this radical alternative in programming languages and method.

In numerical programming, a similar tension between mainstream thinking and the radicals also exists. In the former, a computation is typically a series of successive approximations, which halts when two consecutive approximations differ by a sufficiently small amount. This amount is then used as the error estimate. Of course sophisticated error analyses can be made to suggest more certain knowledge. But such analyses are usually time-consuming and valid only asymptotically. In practice one does not know whether one is close enough to the true value for the asymptotic analysis to be applicable.

The radical alternative in numerical computation is represented by interval methods, where the ideal is to be *sure* that the true value is contained in an interval. It is then the purpose of iteration to shrink such an interval until it is small enough. Here again the goal is *certainty* of knowledge.

We bring these two radical streams together in the research reported in this paper. Both streams are, in their present form, deficient. Logic programming lacks control of numerical errors. Interval methods rely on conventional algorithmic languages and hence lack computation as deduction. *We show that the two can be combined in such a way that rigorously justified claims can be made about the error in numerical computation even if conventional floating-point arithmetic is used.*

**Problem statement.** Many applications involve numerical computations. Thus arithmetic facilities constitute an important component in a programming language. Logic programming languages are no exception. Unfortunately, none of the existing logic programming languages provide arithmetic facilities that are coherent with the relational and deductive paradigm. A major culprit is the occurrence of round-off errors in floating-point arithmetic, which invalidates numerical computations as deduction. In this paper, we consider the CHIP [11] programming language. Through the finite domain concept, CHIP supports a relational form of integer arithmetic. It also has a rational constraint solver, allowing linear equalities, inequalities, and disequalities. The floating-point arithmetic of CHIP, however, is as rudimentary as that of Prolog. The aim of this paper is to improve the arithmetic of CHIP so that its computation steps are valid inferences even when the usual floating-point arithmetic is used.

**Solution.** Our solution consists of three parts. First, interval arithmetic, introduced by Moore [28], helps us to tackle the round-off error problem. Instead of operating on individual floating-point numbers, interval arithmetic manipulates intervals. Interval operations and outward-rounding guarantee the inclusion of solution in the answer interval and ensure the soundness of computation.

Second, traditional interval arithmetic is functional and has been embedded in functional or imperative languages. To develop the required relational version, we use an interval narrowing operation based on work by Cleary [5], which is implemented in BNR-Prolog [30], and similar to the one used by Sidebottom and Havens [34].

Finally, we extend the domain concept to include interval domains and show that the interval narrowing operation is an instance of the Looking-Ahead Inference Rule (LAIR) [39], an inference rule used in the proof procedure of CHIP [11]. Thus interval narrowing becomes a logical inference and numerical computation is deduction.

The paper is organized as follows. Section 2 discusses related work. Relational interval arithmetic, which consists of interval narrowing and a relaxation algorithm, is presented in section 3. In section 4, we describe ICHIP, which is CHIP extended with relational interval arithmetic. We suggest further work in section 5.

# 2 Related Work

We review the role of interval arithmetic in the imperative (or procedural) and the relational programming paradigms. We also discuss proposals that incorporate interval arithmetic into logic programming systems.

## 2.1 Interval Arithmetic

Accounts of conventional interval arithmetic can be found in [28, 1]. Recent trends are summarized in [29].

Interval arithmetic contributes methods guaranteeing correctness at the level of a simple arithmetic expression. Although various algebraic and numerical techniques have been developed, we need a programming language that supports interval arithmetic to realize these techniques into computer programs. As embedded in imperative languages, however, interval arithmetic lacks verification of an entire algorithm involving conditional statements and iterations. This difficulty is caused by the mixing of logic information (*what* the problem is) and control information (*how* the problem is solved) in the imperative programming paradigm. Numerical analysts, such as Kirchner and Kulisch [19, page 37] and Rump [33, page 109], use declarative and mathematical statements of numerical problems to prove correctness of the results of programs. This work shares the goal of logic programming. Thus interval arithmetic is complementary to existing implementations of logic programming.

Bundy [3] recognized the importance of sound arithmetic and implemented a functional interval arithmetic package in Prolog, which is part of a system for checking the conditions of rewrite rules and the solutions to equations in an algebraic manipulation program.

## 2.2 Constraint Interval Reasoning

Research in a relational form of interval arithmetic stems from constraint propagation techniques. A constraint network consists of *nodes*, representing individual parameters having a particular value (known or unknown), connected by *constraints*, which are relations. We attach a *label*, the set of possible values for a node, to each node in the network. Constraint propagation is a process that deduces information from a local group of constraints in a network and propagates the information to the rest of the network. A particular kind of constraint propagation is *label inference*, which uses the constraints to restrict the label. Constraint interval reasoning is a form of label inference, where the labels attached to the nodes of a constraint network are intervals. Davis [7] gives a survey of this topic.

The following are special-purpose systems that use interval label inference. ENVISION [8] performs qualitative reasoning about the behavior of physical systems over time. TMM [9] is a temporal constraint system that records and reasons with changes to the world. EMPRESS-A [42] is a temporal reasoning system with lazy evaluation for solving scheduling problems. SPAM [26] performs spatial reasoning. These systems are based on consistency techniques [23] that handle static constraint networks. To generate constraints dynamically during execution, the described systems are equipped with programming languages tailored to the application. A typical language of this kind is not coherent, having separate sub-languages to describe the constraints, the queries, the answers, the nodes in the network, and the labels. TP [15] is a scheme for constraint reasoning on interval arithmetic. It only considers closed intervals but has an approximate representation of open intervals. Its language is similar to LISP.

Although the above systems use interval arithmetic, they do not take into account in their theoretical basis the effect of outward rounding.

## 2.3 Constraint Logic Programming

There are several proposals to incorporate interval arithmetic into logic programming systems.

**Logical Arithmetic.** Cleary [5] incorporates "logical arithmetic," a relational version of interval arithmetic, into Prolog. He introduces a new term "interval," which requires an extension of the unification algorithm. Cleary presents several "squeezing" algorithms that reduce arithmetic constraints over intervals. A constraint relaxation cycle coordinates the execution of the squeezing algorithms. However, there is a semantic problem in this approach. Variables bound to intervals, which are terms in the Herbrand universe, are re-bound to smaller intervals. This is not part of resolution, where a variable can be bound only once. It is not clear in what other, if any, sense this may be a logical inference. Outward-rounding is implemented but its properties are not studied.

**BNR Prolog.** BNR Prolog [30, 31] provides relational interval arithmetic in a way that is loosely based on Cleary's pre-publication ideas, differing somewhat in particulars. BNR Prolog provides a large set of interval primitives, including some powerful ones that have no conventional counterparts. It has been usefully applied to some sizeable problems such as critical path scheduling [30] and X-ray diffraction crystallography [30, 32].

**Echidna.** Sidebottom and Havens [35] designed and implemented a version of relational interval arithmetic for the Echidna constraint reasoning system [13]. It is based on hierarchical consistency techniques [24] and can handle unions of disjoint intervals. The direct representation of unions of disjoint intervals avoids situations that otherwise require visiting each disjoint interval in turn by backtracking search. Maintaining and traversing this hierarchical data structure incurs space and time overhead. An analysis of the tradeoffs between backtracking and the space and time overhead remains to be done. The hierarchical consistency algorithm used in Echidna is *partial* in the sense of PLAIR [39]. This is because (1) outward rounding computes a larger interval than that specified by the hierarchical consistency algorithm and (2) unions of disjoint intervals are only approximated by a hierarchical data structure. In addition, it is not clear whether Echidna, being a hybrid system of object-oriented and rule-based constraint programming, fits into the constraint logic programming framework. Our method builds on a constraint logic programming language with established semantics.

# 3 Relational Interval Arithmetic

Cleary [5] describes several algorithms to reduce constraints on intervals. These algorithms work under a basic principle: they narrow intervals associated with a constraint by removing values that do not satisfy the constraint. We study the set-theoretic aspect of the algorithms and generalize them for narrowing intervals constrained by any relation $p$ on $\mathbb{R}^n$ satisfying certain criterion. We then discuss interval narrowing for some common arithmetic relations. Interval narrowing is designed for the reduction of a single constraint. Typically, several constraints interact with one another by sharing intervals, resulting in a constraint *network*. We present an algorithm that coordinates the applications of interval narrowing to constraints in a network.

## 3.1 Basics of Interval Arithmetic

A good introduction to interval arithmetic can be found in [1]. We use $\mathbb{R}$ to denote the set of real numbers and $\mathbb{F}$ a set of floating-point numbers. For the purpose of this paper, it suffices to assume that $\mathbb{F}$ is any finite subset of $\mathbb{R}$. If $a, b \in \mathbb{F}$ and $a < b$, then $a$ and $b$ are *adjacent* if there does not exist a $c \in \mathbb{F}$ such $a < c < b$. To represent intervals, we use the usual mathematical notations, such as $(1, 2]$. Intuitively, an interval is a segment, possibly infinite, of the real line. To represent intervals without a lower or an upper bound, we use as bound the symbols $-\infty$ or $+\infty$ respectively. Note that $-\infty$ and $+\infty$ can only be used with open bounds since they are not members of $\mathbb{R}$. To be precise, we define the set of *real intervals*, $I(\mathbb{R})$, by

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

In the above definition, "$a$" is the *lower* bound and "$b$" is the *upper* bound. "[" and "]" are used to denote *closed* bounds; and "(" and ")" are used to denote *open* bounds. Traditional interval arithmetic only considers closed intervals. Our framework uses also open and mixed intervals so as to obtain a precise set-theoretic characterization of the interval narrowing operation. This characterization is essential in connecting relational interval arithmetic with the operational semantics of CHIP to be discussed in section 4.

We distinguish between real intervals and *floating-point intervals*, whose bounds are floating-point numbers. Replacing $\mathbb{R}$ by $\mathbb{F}$ in the definition of $I(\mathbb{R})$, we obtain the definition of floating-point intervals $I(\mathbb{F})$. Note that real and floating-point intervals differ only in the bounds but every interval, real or floating-point, denotes a set of real numbers. For example, $[e, \pi) = \{x \mid e \leq x < \pi\}$, $(-\infty, 4.5] = \{x \mid x \leq 4.5\}$, and $(-\infty, +\infty) = \mathbb{R}$. An interval can also be empty, as in $(4, 1] = \emptyset$. We impose a partial ordering on real intervals; an interval $I_1$ is *smaller than or equal to* an interval $I_2$ if and only if $I_1 \subseteq I_2$. Let $\mathcal{I}$ be a set of intervals ($\mathcal{I} \subseteq I(\mathbb{R})$ or $\mathcal{I} \subseteq I(\mathbb{F})$). $I \in \mathcal{I}$ is the *smallest* interval in $\mathcal{I}$ if $I$ is smaller than or equal to $I'$ for all $I' \in \mathcal{I}$.

Floating-point intervals are not closed under interval operations. In floating-point arithmetic, real numbers are approximated by floating-point numbers using rounding or truncation. In interval arithmetic, we approximate real intervals by floating-point intervals using the *outward-rounding* function, $\xi : I(\mathbb{R}) \rightarrow I(\mathbb{F})$; if $J$ is a non-empty real interval,

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

The introduction of outward-rounding complicates the proofs of some seemingly simple facts about relational interval arithmetic.

**Lemma 3.1**: If $I$ is a non-empty real interval, then $\xi(I)$ is the smallest floating-point interval containing $I$.

**Proof:** Floating-point intervals are closed under the set intersection operation. Therefore, $\xi(I)$ is a floating-point interval. In addition, $I \subseteq \xi(I)$ since $\xi(I)$ is the intersection of all floating-point intervals containing $I$. Also, if $I \subseteq I'$ for some $I' \in I(\mathbb{F})$, then $\xi(I) \subseteq I'$ by the definition of $\xi$. ■

Lemma 3.1 can be the basis of an implementation of $\xi$. $\mathbb{R}$ is a continuum and $\mathbb{F}$ consists of floating-point numbers scattered along the real line. Intuitively, we round to the "left" at the lower bound and to the "right" at the upper bound of $I$.

Lemma 3.1 implies that $\xi(I) = I$ for all floating-point intervals $I$. Thus, $\xi$ is *idempotent*, i.e. $\xi(\xi(I)) = \xi(I)$ for any interval $I$. In the following, we discuss some properties of $\xi$. The next lemma shows that outward-rounding is monotone.

**Lemma 3.2**: Let $I_1, I_2 \in I(\mathbb{R})$. If $I_1 \subseteq I_2$, then $\xi(I_1) \subseteq \xi(I_2)$.

**Proof:** By lemma 3.1, $I_1 \subseteq \xi(I_1)$ and $I_1 \subseteq I_2 \subseteq \xi(I_2)$. Since $\xi(I_1)$ is the smallest floating-point interval containing $I_1$ by lemma 3.1, $\xi(I_1) \subseteq \xi(I_2)$. ■

We state without proof the following results, which show that every element of $\xi(I)$ is contained in a neighborhood of $I$.

**Lemma 3.3**: If $I \in I(\mathbb{R})$, then $\xi(I) = \bigcup\{\xi([x,x]) \mid x \in I\}$. ■

**Lemma 3.4**: If $I \in I(\mathbb{F})$ and $x \in I$, then $\xi([x,x]) \subseteq \xi(I) = I$. ■

Cleary [5, 18] gives detailed algorithms to compute the types of the resulting bounds of an interval function application. Each algorithm is of constant overhead for a particular interval function. In practice, the type of the bounds is not very informative, especially in the presence of outward-rounding. Therefore, it is reasonable to choose an implementation with only closed intervals, as suggested by Davis [7] and adopted in BNR Prolog, to avoid this overhead.

## 3.2 Interval Narrowing

An *interval constraint*, which will be referred to as *constraint* hereafter, is of the form $(p, \vec{I})$, where $p$ is a relation on $\mathbb{R}^n$ and $\vec{I} = \langle I_1, \ldots, I_n \rangle$ is a sequence of floating-point intervals. The constraint $(p, \vec{I})$ means that

$$\exists X_i \in I_i (i = 1, \ldots, n) \text{ such that } p(X_1, \ldots, X_n).$$

Note that the number of intervals in the sequence $\vec{I}$ is equal to the arity of $p$. For example, the constraint $(\texttt{add}, \langle [1,3], [2,4], [4,6] \rangle)$ means that

$$\exists X \in [1,3], \exists Y \in [2,4], \exists Z \in [4,6] \text{ such that } \texttt{add}(X, Y, Z).$$

For any relation $p$ of arity $n$, we can associate $n$ set-valued functions with $p$:

$$\begin{aligned}
& F_i(p)(S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_n) \\
= \; & \{s_i \,|\, (s_1, \ldots, s_n) \in ((S_1 \times \cdots \times S_{i-1} \times \pi_i(p) \times S_{i+1} \times \cdots \times S_n) \bigcap p)\} \\
= \; & \pi_i((S_1 \times \cdots \times S_{i-1} \times \pi_i(p) \times S_{i+1} \times \cdots \times S_n) \bigcap p),
\end{aligned}$$

where $i = 1, \ldots, n$, the $S_i$'s are sets, and $\pi_i$ is the projection function defined by

$$\pi_i(p) = \{s_i \,|\, (s_1, \ldots, s_n) \in p\}.$$

Intuitively, $F_i(p)(S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_n)$ is the set of values for the $i^{\text{th}}$ argument of $p$ if we restrict the values of the other arguments to $S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_n$ respectively. We use $F_i(p)$ as the basis of interval functions. In interval arithmetic, it is essential that the results of interval operations are intervals. Therefore, we consider only relations $p$ on $\mathbb{R}^n$, such that each of its $F_i(p)$ maps intervals to intervals. For example, the relation $\texttt{add} = \{(x, y, z) \,|\, x, y, z \in \mathbb{R}, x + y = z\}$ satisfies this requirement. We have

$$F_1(\texttt{add})(I_2, I_3) = I_3 \ominus I_2, \qquad F_2(\texttt{add})(I_1, I_3) = I_3 \ominus I_1, \qquad F_3(\texttt{add})(I_1, I_2) = I_1 \oplus I_2,$$

where $A \oplus B = \{a + b \,|\, a \in A, b \in B\}$ and $A \ominus B = \{a - b \,|\, a \in A, b \in B\}$. By the continuity of the $+$ and $-$ operators, $A \oplus B$ and $A \ominus B$ are intervals if $A$ and $B$ are intervals.

We now specify *interval narrowing* as a collection of input-output pairs.

**Input:** $\vec{I} = \langle I_1, \ldots, I_n \rangle$, where $I_i$ is a floating-point interval.

**Output:** $\vec{I}' = \langle I'_1, \ldots, I'_n \rangle$, where $I'_i = I_i \bigcap \xi(F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n))$.

The application of $\xi$ in the formula ensures that the output intervals are floating-point intervals. If one or more $I'_i$ is empty, then interval narrowing *fails* and the constraint $(p, \vec{I})$ is *inconsistent*. Otherwise it succeeds with $I'_1, \ldots, I'_n$ as output. Note that the output interval $I'_i$ is a subset of the corresponding input interval $I_i$.

It is important that interval narrowing does not eliminate values that can satisfy the constraint. The following theorem guarantees this property.

**Theorem 3.5**: Let $C$ be $(p, \langle I_1, \ldots, I_n \rangle)$. If $(x_1, \ldots, x_n) \in p$ and $\langle I'_1, \ldots, I'_n \rangle$ are the output intervals obtained from interval narrowing of $C$, then $(x_1, \ldots, x_n) \in I_1 \times \cdots \times I_n$ if and only if $(x_1, \ldots, x_n) \in I'_1 \times \cdots \times I'_n$.

**Proof:** Since $I'_1 \times \cdots \times I'_n \subseteq I_1 \times \cdots \times I_n$, the if-part of the lemma is true. We prove the only-if-part of the lemma. Suppose $(x_1, \ldots, x_n) \in I_1 \times \cdots \times I_n \cap p$. We have $x_i \in I_i$ for $i = 1, \ldots, n$ and $x_i \in F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)$ by definition. Therefore $x_i \in I'_i$ and $(x_1, \ldots, x_n) \in I'_1 \times \cdots \times I'_n$. ■

The next lemma assists in expressing interval narrowing in terms of relational operations.

**Lemma 3.6**: For $A \in I(\mathbb{F})$ and $B \in I(\mathbb{R})$, $A \bigcap \xi(B) = \xi(A \bigcap B)$. ■

We rewrite the output of interval narrowing as follows:

**Theorem 3.7**: $I'_i = \xi(\pi_i((I_1 \times \cdots \times I_n) \bigcap p))$.

**Proof:**

$$
\begin{aligned}
I_i' &= I_i \bigcap \xi(F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)) \\
&= \xi(I_i \bigcap F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)) \text{ by lemma 3.6} \\
&= \xi(I_i \bigcap \pi_i((I_1 \times \cdots \times I_{i-1} \times \pi_i(p) \times I_{i+1} \times \cdots \times I_n) \bigcap p) \\
&= \xi(I_i \bigcap \{x_i \,|\, (x_1, \ldots, x_n) \in ((I_1 \times \cdots \times I_{i-1} \times \pi_i(p) \times I_{i+1} \times \cdots \times I_n) \bigcap p))\} \\
&= \xi(\{x_i \in I_i \,|\, (x_1, \ldots, x_n) \in ((I_1 \times \cdots \times I_{i-1} \times \pi_i(p) \times I_{i+1} \times \cdots \times I_n) \bigcap p))\} \\
&= \xi(\{x_i \,|\, (x_1, \ldots, x_n) \in ((I_1 \times \cdots \times I_n) \bigcap p)\}) \\
&= \xi(\pi_i((I_1 \times \cdots \times I_n) \bigcap p))
\end{aligned}
$$

■

In essence, interval narrowing computes the intersection of $I_1 \times \cdots \times I_n$ and $p$, and outward-rounds each projection of the resulting relation. Moreover, $I_1' \times \cdots \times I_n'$ is the *cylindrical closure* [2] of order 1 of $(I_1 \times \cdots \times I_n) \bigcap p$, augmented with outward-rounding. Figure 1 illustrates the interval narrowing of the constraint $(\mathtt{le}, \langle I_1, I_2 \rangle)$, where $\mathtt{le} = \{(x, y) \,|\, x, y \in \mathbb{R}, x \leq y\}$. In the diagram, the initial floating-point intervals are $I_1$ and $I_2$. The dotted region denotes the relation `le`; the region for $I_1 \times I_2$ is shaded with a straight-line pattern. Interval narrowing returns $I_1'$ and $I_2'$ by taking the projections of the intersection of the two regions. There is no need to perform outward-rounding in this example since the bounds of $I_1'$ and $I_2'$ share those of $I_1$ and $I_2$.

With the alternative definition, we prove an important property of interval narrowing: *monotonicity*.

**Lemma 3.8**: Let $\vec{I} = \langle I_1, \ldots, I_n \rangle$, $\vec{I}' = \langle I_1', \ldots, I_n' \rangle$, $\vec{J} = \langle J_1, \ldots, J_n \rangle$, and $\vec{J}' = \langle J_1', \ldots, J_n' \rangle$ be tuples of floating-point intervals, and $p$ a relation in $\mathbb{R}^n$, where $I_k \subseteq J_k$ for $k = 1, \ldots, n$. If, by interval narrowing, $\vec{I}'$ is obtained from $\vec{I}$ and $\vec{J}'$ is obtained from $\vec{J}$, then $I_k' \subseteq J_k'$ for $k = 1, \ldots, n$.

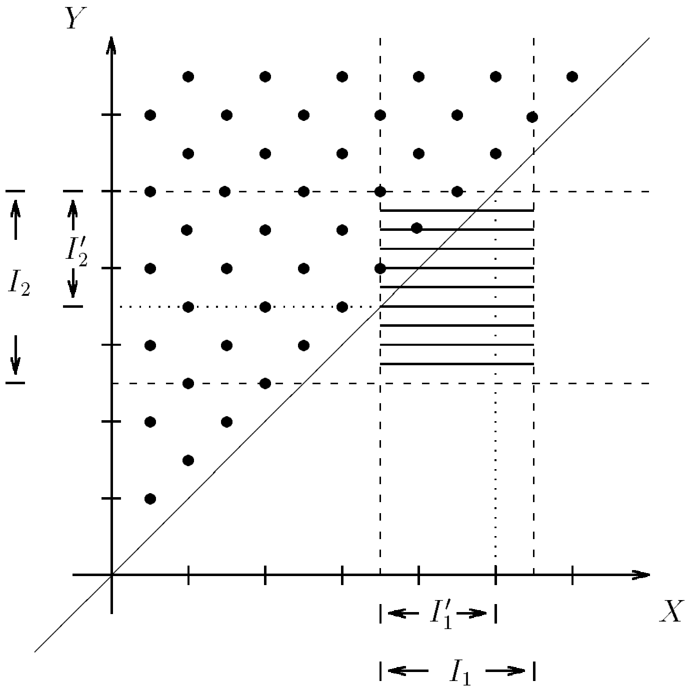


Figure 1: Illustration of interval narrowing.

**Proof:** Let $1 \leq i \leq n$.

$$
\begin{aligned}
& I_k \subseteq J_k \text{ for } k = 1, \ldots, n \\
\Rightarrow\ & I_1 \times \cdots \times I_n \subseteq J_1 \times \cdots \times J_n \\
\Rightarrow\ & (I_1 \times \cdots \times I_n) \cap p \subseteq (J_1 \times \cdots \times J_n) \cap p \\
\Rightarrow\ & \pi_i((I_1 \times \cdots \times I_n) \cap p) \subseteq \pi_i((J_1 \times \cdots \times J_n) \cap p) \\
\Rightarrow\ & \xi(\pi_i((I_1 \times \cdots \times I_n) \cap p)) \subseteq \xi(\pi_i((J_1 \times \cdots \times J_n) \cap p)) \text{ by lemma 3.2} \\
\Rightarrow\ & I'_i \subseteq J'_i \text{ by theorem 3.7}
\end{aligned}
$$

■

### 3.3 Arithmetic Primitives

A relational interval arithmetic system should support primitive arithmetic constraints, such as addition and multiplication. More complex constraints can then be built from these primitives. To ensure that a relation $p$ is suitable for interval narrowing, we need to check that each $F_i(p)$ maps from real intervals to real intervals. If $p$ is one of

$$\texttt{eq} = \{(x,x) \mid x \in \mathbb{R}\}, \qquad \texttt{le} = \{(x,y) \mid x, y \in \mathbb{R}, x \leq y\},$$
$$\texttt{lt} = \{(x,y) \mid x, y \in \mathbb{R}, x < y\}, \qquad \texttt{add} = \{(x,y,z) \mid x, y, z \in \mathbb{R}, x + y = z\},$$

we can verify easily that the $F_i(p)$'s satisfy the criterion.

The multiplication relation, $\texttt{mult} = \{(x,y,z) \mid x, y, z \in \mathbb{R}, xy = z\}$, requires further explanation. Consider

$$F_1(\texttt{mult})(I_2, I_3) = I_3 \oslash I_2 \qquad \text{and} \qquad F_2(\texttt{mult})(I_1, I_3) = I_3 \oslash I_1,$$

where $A \oslash B = \{a/b \mid a \in A, b \in B, b \neq 0\}$. Suppose that $A$ and $B$ are intervals. The problem is that $A \oslash B$ may not be an interval. But the following property may easily be verified: If $A$ contains 0 as an interior point or $B$ does not, then $A \oslash B$ is an interval. Otherwise (if $A$ does not contain 0 as interior point and $B$ does), $A \oslash B$ is the union of two disjoint intervals, one containing only positive reals and the other only negative ones.

As suggested in [5], we can circumvent the problem by partitioning `mult` into $\texttt{mult}^+$ and $\texttt{mult}^-$, where

$$\begin{aligned} \texttt{mult}^+ &= \{(x,y,z) \mid x, y, z \in \mathbb{R}, x \geq 0, xy = z\}, \\ \texttt{mult}^- &= \{(x,y,z) \mid x, y, z \in \mathbb{R}, x < 0, xy = z\}. \end{aligned}$$

As will be seen, the functions $F_i(\texttt{mult}^+)$ and $F_i(\texttt{mult}^-)$, with $i = 1, 2,$ or 3 do map intervals to intervals. By restricting interval narrowing to one partition or the other, we can guarantee that the result of interval division is an interval. When a `mult` constraint is encountered, we choose one of the partitions and perform narrowing; the other partition is visited upon automatic backtracking or under user control.

Because $\otimes$ maps intervals to intervals, so do $F_3(\texttt{mult}^+)(I_1, I_2) = (I_1 \cap \mathbb{R}^+) \otimes I_2$, where $\mathbb{R}^+$ is the nonnegative part of $\mathbb{R}$. By the above property of $\oslash$, $F_2(\texttt{mult}^+)(I_1, I_3) = I_3 \oslash (I_1 \cap \mathbb{R}^+)$ is an interval. This is the more obvious consequence of the property, because the definition of $\texttt{mult}^+$ requires the use of $I_1 \cap \mathbb{R}^+$, of which 0 is not an interior point.

The property also guarantees that $I_3 \oslash I_2$ is a single interval or a disjoint union of two intervals, one consisting of positive reals only and the other of negative reals only. Hence $F_1(\texttt{mult}^+)(I_2, I_3) = (I_3 \oslash I_2) \cap \mathbb{R}^+$ is either a single interval, or is empty.

It can be shown in a similar way that the corresponding functions for $\texttt{mult}^-$ map intervals to intervals.

An advantage of relational interval arithmetic is that we do not have the division-by-zero problem. In most computer arithmetic systems, an attempt to compute, say "$X = Y/0$," will lead to some form of exception being raised. In relational notation, the last equation becomes "$\texttt{mult}(X, 0, Y)$." For example, the constraint $(\texttt{mult}^+, \langle(4, +\infty), [0, 0], [-3, 5)\rangle)$ can be reduced by interval narrowing as follows:

| Input Intervals $= \langle I_1, I_2, I_3\rangle$ | $(4, +\infty)$ | $[0, 0]$ | $[-3, 5)$ |
|---|---|---|---|
| $\xi(F_1(\texttt{mult}^+)(I_2, I_3)) = \xi(([-3, 5) \oslash [0, 0]) \cap \mathbb{R}^+)$ | $[0, +\infty)$ | | |
| $\xi(F_2(\texttt{mult}^+)(I_1, I_3)) = \xi([-3, 5) \oslash (4, +\infty))$ | | $(-0.75, 1.25)$ | |
| $\xi(F_3(\texttt{mult}^+)(I_1, I_2)) = \xi(([0, 0]) \otimes (4, +\infty))$ | | | $[0, 0]$ |
| Output Intervals $= \langle I_1', I_2', I_3'\rangle$ | $(4, +\infty)$ | $[0, 0]$ | $[0, 0]$ |

The idea is that the first argument of $\texttt{mult}^+$ can take on any real value and multiplying it by zero always yield zero. The reduced constraint is $(\texttt{mult}^+, \langle(4, +\infty), [0, 0], [0, 0]\rangle)$.

Relations induced from transcendental functions and the disequality relation, such as $\texttt{sin} = \{(x, y) \mid x, y \in \mathbb{R}, y = \sin(x)\}$ and $\texttt{dif} = \{(x, y) \mid x, y \in \mathbb{R}, x \neq y\}$, also suffer from the same problem as the $\texttt{mult}$ relation. Similarly, we apply the partitioning technique to the relations. However, there are infinitely many partitions for $\texttt{sin}$, $\texttt{cos}$, and $\texttt{tan}$. In practice, we restrict the use of $\texttt{sin}$ to the partitions in $-\pi/2$ to $3\pi/2$, $\texttt{cos}$ to those in $-\pi$ to $\pi$, and $\texttt{tan}$ to those in $-\pi/2$ to $\pi/2$.

The partitioning scheme creates backtracking points. This fits into the constraint logic programming framework. There can be multiple solutions to a problem, some of which may be duplicates of one another. Existing constraint logic programming systems exhibit the same behaviour.

## 3.4 Constraint Networks

The interval narrowing operation discussed so far reduces individual constraints. In practice, we have more than one constraint in a problem. These constraints may depend on one another by sharing intervals. By naming an interval by a variable and by having a variable occur in more than one constraint, we indicate that constraints share intervals. Note that the material in this section is not related to logic programming but is in conventional notation with destructive assignment.

We define an *interval network*, which will be referred to as a *network* hereafter, to be a set of constraints. Consider the quadratic equation $v(v-1) = 6$. Suppose our initial guess for the positive root of the equation is $[1, 100]$. We can express the equation by the following network:

$$\{(\mathtt{add}, \langle V_1, [1,1], V\rangle), (\mathtt{mult}^+, \langle V, V_1, [6,6]\rangle)\},$$

where the variables $V$ and $V_1$ are initially assigned intervals $[1, 100]$ and $(-\infty, +\infty)$ respectively.

Before we present the reduction of networks, we discuss the following two observations. First, the reduction of a constraint $C$ in the network affects other constraints that share variables with $C$. Second, interval narrowing is *idempotent* as shown in the following lemma.

**Lemma 3.9**: Let $\vec{I} = \langle I_1, \ldots, I_n\rangle$, $\vec{I'} = \langle I'_1, \ldots, I'_n\rangle$, and $\vec{I''} = \langle I''_1, \ldots, I''_n\rangle$ be sequences of floating-point intervals and $p$ a relation on $\mathbb{R}^n$. If, by interval narrowing, $\vec{I'}$ is obtained from $\vec{I}$ and $\vec{I''}$ is obtained from $\vec{I'}$, then $\vec{I'} = \vec{I''}$.

**Proof:** To prove the equality of $\vec{I}'$ and $\vec{I}''$, we prove $I'_i = I''_i$ for $i = 1, \ldots, n$. By the definition of interval narrowing and lemma 3.6, we have

$$I'_i = \xi(I_i \bigcap F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n)),$$
$$I''_i = \xi(I'_i \bigcap F_i(p)(I'_1, \ldots, I'_{i-1}, I'_{i+1}, \ldots, I'_n)).$$

It is obvious that $I''_i \subseteq I'_i$. Next we prove $I'_i \subseteq I''_i$.

Suppose $a'_i \in I'_i$. There exists $a_i \in (I_i \bigcap F_i(p)(I_1, \ldots, I_{i-1}, I_{i+1}, \ldots, I_n))$ such that $a'_i \in \xi([a_i, a_i])$ by lemma 3.3. By the definition of $F_i(p)$, for $j = 1, \ldots, i-1, i+1, \ldots, n$, there exists $a_j \in I_j$ such that $(a_1, \ldots, a_n) \in p$. Since $a_i \in I_i$, we have $a_j \in I'_j$ for each $j$. Thus, $a_i \in F_i(p)(I'_1, \ldots, I'_{i-1}, I'_{i+1}, \ldots, I'_n))$. This implies that $a_i \in I''_i$. By lemma 3.4, $a'_i \in I''_i$. ■

A constraint $(p, \vec{I})$ is *stable* if applying interval narrowing on $\vec{I}$ results in $\vec{I}$. A network is *stable* if every constraint in the network is stable. The reduction of a network amounts to transforming the network into a stable one.

A naive approach for network reduction is to reduce each constraint in the network in round-robin fashion until every constraint becomes stable. As suggested by lemma 3.9, this method is inefficient since much computation is wasted in reducing stable constraints. Algorithm 1, which is based on the constraint relaxation algorithm described in [5], is the pseudocode of a more efficient procedure. It is similar to the Waltz algorithm [41] and the arc consistency algorithm AC-3 [23] in that it tries to avoid reducing stable constraints. Without loss of generality, we assume that every constraint in the network is of the form $(p, \langle V_1, \ldots, V_n \rangle)$, where the $V_i$'s are interval-valued variables.

The idea is to maintain the constraints of the network in two lists: the active list $A$ and the passive list $P$. The active list contains constraints that are possibly unstable; whereas the passive list contains stable constraints. Initially, the passive list $P$ is empty and the active list $A$ contains all the constraints in the network. Then each constraint in $A$ is reduced in turn using interval narrowing. If any of the intervals are empty after narrowing, the algorithm exits with failure. We say that the network is *inconsistent*. If any interval is narrowed, the

initialize $A$ to the list of all constraints in the network
initialize $P$ to the empty list
**while** $A$ is not empty
  remove the first constraint, $(p, \vec{V})$, from $A$
  apply interval narrowing on $(p, \vec{V})$ to obtain $\vec{V}'$
  **if** interval narrowing fails **then**
    exit with failure
  **else if** $\vec{V} \neq \vec{V}'$ **then**
    $\vec{V} \leftarrow \vec{V}'$
    **foreach** constraint $(q, \vec{Y})$ in $P$
      **if** $\vec{V}$ and $\vec{Y}$ share narrowed variable(s) **then**
        remove $(q, \vec{Y})$ from $P$ and append it to $A$
      **endif**
    **endforeach**
  **endif**
  append $(p, \vec{V})$ to the end of $P$
**endwhile**

Algorithm 1: A Relaxation Algorithm.

algorithm updates the variables. Moreover, constraints in $P$ that share narrowed variables with the reduced constraint may become unstable. The algorithm promotes these constraints from $P$ to $A$. The reduced constraint is stable and is appended to the end of $P$. This process repeats until $A$ is empty.

Algorithm 1 resembles a classical iterative numerical-approximation technique called "relaxation" [36, 37]. Numerical relaxation may have numerical stability problems; the procedure may fail to converge or terminate even when the constraints have a solution. Algorithm 1 does not suffer from this problem as shown in the following theorem.

**Theorem 3.10**: Algorithm 1 always terminates. The resulting network is either inconsistent or stable.

**Proof:** Algorithm 1 halts either when interval narrowing of a constraint fails or the list $A$

becomes empty. In the former case, the network is inconsistent. For the latter case, we observe that the size of list $A$ decreases after each iteration of the algorithm unless variables are narrowed and constraints are moved from $P$ to $A$. However, the precision of a floating-point system is finite and thus interval narrowing cannot occur indefinitely due to the use of outward-rounding. Therefore, list $A$ must become empty after a finite number of iterations. All constraints of the network are now in $P$ and are stable. Thus the network is stable. ■

In the following, we show how algorithm 1 finds the positive root of the equation $v(v-1)-6=0$ with initial guess $v \in [1,100]$. Initially, the active list of constraints, $A$, contains both constraints

$$A = [C_1, C_2] = [(\texttt{add}, \langle V_1, [1,1], V\rangle), (\texttt{mult}^+, \langle V, V_1, [6,6]\rangle)],$$

and the passive list, $P$, is empty, where $V = [1,100]$ and $V_1 = (-\infty, +\infty)$. We remove the first constraint $(\texttt{add}, \langle V_1, [1,1], V\rangle)$ from $A$ and reduce it by interval narrowing as follows:

| Input Intervals = $\langle I_1, I_2, I_3\rangle$ | $(-\infty,+\infty)$ | $[1,1]$ | $[1,100]$ |
|---|---|---|---|
| $\xi(F_1(\texttt{add})(I_2, I_3)) = \xi([1,100] \ominus [1,1])$ | $[0,99]$ | | |
| $\xi(F_2(\texttt{add})(I_1, I_3)) = \xi([1,100] \ominus (-\infty,+\infty))$ | | $(-\infty,+\infty)$ | |
| $\xi(F_3(\texttt{add})(I_1, I_2)) = \xi((-\infty,+\infty) \oplus [1,1])$ | | | $(-\infty,+\infty)$ |
| Output Intervals = $\langle I_1', I_2', I_3'\rangle$ | $[0,99]$ | $[1,1]$ | $[1,100]$ |

The updated values of $V$ and $V_1$ are $[1,100]$ and $[0,99]$ respectively. Similar narrowing is performed on the $\texttt{mult}^+$ constraint. This process repeats until the precision of the underlying floating-point system is reached and no more narrowing takes place. In this example, algorithm 1 takes 42 steps to reach the stable state. The history of the values of $A$, $P$, $V$, and $V_1$, with four significant figures, after each narrowing step is summarized in table 1.

# 4 Extending CHIP with Interval Arithmetic

So far, we have explained how a *static* network of constraints in terms of floating-point intervals can be made stable using algorithm 1. We have not considered how such networks

| $A$ | $P$ | $V$ | $V_1$ |
|---|---|---|---|
| $[C_1, C_2]$ | $[]$ | $[1, 100]$ | $(-\infty, +\infty)$ |
| $[C_2]$ | $[C_1]$ | $[1, 100]$ | $[0, 99]$ |
| $[C_1]$ | $[C_2]$ | $[1, 100]$ | $[0.06, 6]$ |
| $[C_2]$ | $[C_1]$ | $[1.06, 7]$ | $[0.06, 6]$ |
| $[C_1]$ | $[C_2]$ | $[1.06, 7]$ | $(0.8571, 5.661)$ |
| $[C_2]$ | $[C_1]$ | $(1.857, 6.661)$ | $(0.8571, 5.661)$ |
| $[C_1]$ | $[C_2]$ | $(1.857, 6.661)$ | $(0.9009, 3.231)$ |
| $[C_2]$ | $[C_1]$ | $(1.900, 4.231)$ | $(0.9009, 3.231)$ |
| $[C_1]$ | $[C_2]$ | $(1.900, 4.231)$ | $(1.418, 3.157)$ |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $[]$ | $[C_1, C_2]$ | $(2.999, 3.001)$ | $(1.999, 2.001)$ |

Table 1: Traces of $A$, $P$, $V$, and $V_1$.

can be specified. The number of constraints and variables in a network can be large. This makes the specification a tedious and error-prone task. We look for a programming language that can (1) describe constraints, queries, intervals, answers, and variables, in a coherent manner, and (2) describe a network concisely. CHIP [11], based on Horn clauses with domain variables [39] and consistency techniques [23], satisfies those requirements. Both domains and intervals are sets in which the solution, if it exists, must lie. *This suggests that we view intervals as domains in CHIP.* A network can be represented implicitly in a CHIP program and the parts of the network are generated dynamically during derivation.

Domains in CHIP are finite sets of constants. The restriction to finite sets allows a simple implementation based on enumeration. We extend domains to infinite sets that should satisfy the following criteria. First, the infinite sets can be represented by a finite and small amount of resources. Second, there is a terminating algorithm to compute the intersection of these infinite sets, since intersection is an essential operation in the unification of domain variables. Third, there is a terminating algorithm to implement the Looking-Ahead Inference Rule (LAIR) [39] for the infinite domains. It is easy to verify that intervals satisfy the first two criteria. For the third criterion, we show in the following sections that the interval narrowing operation is an implementation of LAIR. Finally, we show that algorithm 1 is captured in the computation rule of CHIP.

## 4.1 An Overview of CHIP

We review concepts of CHIP necessary for discussion in this paper. Interested readers are referred to [38, 39] for detailed treatment of the subject.

The underlying theory of CHIP is based on a first-order theory extended with *domain variables*. A *domain* is a non-empty finite set of constants. *Domain variables* with domain $D$ are superscripted with $D$, as in $X^D$. Intuitively, a domain variable $X^D$ can be bound only to elements of $D$. The usual variables of first-order logic are called *simple variables*. The introduction of the domain variables calls for an extended unification algorithm [39]. The three following cases should be added to the usual algorithm:

1. If a domain variable and a constant have to be unified, the unification succeeds if the constant is in the domain of the domain variable and binds the latter to the former. Otherwise the unification fails.

2. If two domain variables have to be unified, the unification succeeds if the intersection of their domains is non-empty and binds both variables to a new domain variable ranging over this intersection. Otherwise the unification fails.

3. If a domain variable and a simple variable have to be unified, the unification succeeds and binds the simple variable to the domain variable.

SLD-resolution with this extended unification algorithm is called SLDD-resolution. The proofs of the soundness and completeness of SLDD-resolution can be based on those of SLD-resolution [22], which do not take into account the fact that unifications take place in an empty and unsorted equational theory.

The proof procedure for logic programming with domains is based on SLDD-resolution and three new inference rules based on consistency techniques [23]: forward checking, looking ahead, and partial looking ahead. Looking ahead is a generalization of forward checking. Partial looking ahead, which is used to justify the implementation of some built-in predicates, is an approximation of looking ahead. For this paper, it is sufficient to restrict our discussion to the *Looking-Ahead Inference Rule* (LAIR).

We begin by defining the kind of predicates, to which LAIR can be applied. An $n$-ary predicate $p$ is a *domain constraint* if and only if for any ground terms $g_1, \ldots, g_n$, either $p(g_1, \ldots, g_n)$ has a successful refutation or $p(g_1, \ldots, g_n)$ has only finitely failed derivations. In general, it is a semi-decidable problem to decide whether a predicate is a domain constraint. In the CHIP language [11], it is the programmer's responsibility to declare a predicate to be a domain constraint. A domain constraint $p(t_1, \ldots, t_n)$ is *lookahead-checkable* if and only if there exists at least one $t_i$ that is a domain variable, and each of the other arguments is either ground or a domain variable. The domain variables in $t_1, \ldots, t_n$ are called the *lookahead variables*. We have now enough concepts to define LAIR.

Let $P$ be a program and $G_i = \leftarrow A_1, \ldots, A_m, \ldots, A_k$ be a goal. $G_{i+1}$ is *derived by LAIR* from $G_i$ and $P$ using the substitution $\theta_{i+1}$ if the following conditions hold:

1. $A_m$ is lookahead-checkable and $X_1, \ldots, X_n$ are the lookahead variables of $A_m$, which range respectively over $D_1, \ldots, D_n$.

2. For all $1 \leq j \leq n$, $E_j = \{a_j \in D_j \mid \exists a_1 \in D_1, \ldots, a_{j-1} \in D_{j-1}, a_{j+1} \in D_{j+1}, \ldots, a_n \in D_n \text{ such that } P \models A_m\theta\} \neq \emptyset$, where $\theta = \{X_1/a_1, \ldots, X_n/a_n\}$.

3. $z_i$ is the constant $c$ if $E_j = \{c\}$ or a new domain variable ranging over $E_j$ otherwise.

4. $\theta_{i+1}$ is $\{X_1/z_1, \ldots, X_n/z_n\}$.

5. $G_{i+1}$ is either $\leftarrow (A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\theta_{i+1}$ if at most one $z_i$ is a domain variable, or $\leftarrow (A_1, \ldots, A_k)\theta_{i+1}$ otherwise.

A soundness result for the LAIR is presented in [38] but the completeness result does not hold in general. However, a sound and complete proof procedure using the LAIR and SLDD-derivation can still be programmed but it may be necessary to use normal derivation for lookahead-checkable predicates.

In CHIP, the programmer is responsible for specifying the inference rule to be used for atoms with particular predicate names and also the preconditions for its use. A *lookahead declaration* for an $n$-ary predicate $p$ is a unique expression for $p$ of the following form:

lookahead $p(a_1, \ldots, a_n)$,

where each $a_i$ is either '$g$' or '$d$'. All atoms with predicate $p$ are said to be *submitted* to this lookahead declaration. Atoms submitted to lookahead declarations are reduced by LAIR during derivations. A lookahead declaration also specifies the preconditions that an atom has to satisfy before it can be reduced. An atom $A$ submitted to a lookahead declaration is *lookahead-available* if and only if

1. all the arguments of $A$ corresponding to a '$g$' in the lookahead declaration are ground;

2. $A$ is lookahead-checkable.

The *lookahead-efficient computation rule* used in CHIP only selects an atom, submitted to a lookahead declaration, after it becomes lookahead-available.

## 4.2 Approximating an Arithmetic Relation

In logic programming, it is essential to know the relations denoted by the predicates in a program. In Prolog and CLP($\mathcal{R}$) [17], the meaning of the arithmetic predicates is defined in terms of real arithmetic. The floating-point implementations of these predicates, however, are incorrect with respect to the definitions of the predicates. We cannot assume real arithmetic on real computers. The effect of rounding or truncation should be captured in the definition of arithmetic predicates.

Let $a < b$ be two consecutive floating-point numbers. It is easy to verify that for all $x \in (a, b)$, $\xi([x, x]) = (a, b)$. In other words, numbers in $(a, b)$ are indistinguishable under a floating-point system. We define the *approximation* $p^\epsilon$ of a relation $p \subseteq I\!R^n$ by

$$p^\epsilon = \{(x_1, \ldots, x_n) \mid \exists\, (x'_1, \ldots, x'_n) \in p \text{ s.t. } x_i \in \xi([x'_i, x'_i]) \text{ for } i = 1, \ldots, n\}.$$

Intuitively, $p^\epsilon$ is the union of the "neighbourhoods" of all points in $p$.

## 4.3 Interval Narrowing as LAIR

We show that interval narrowing and the Looking-Ahead Inference Rule LAIR perform the same amount of narrowing on the input intervals.

Let $X_1^{I_1}, \ldots, X_n^{I_n}$ be domain variables with interval domains $I_1, \ldots, I_n$. Reducing the constraint $p^\epsilon(X_1^{I_1}, \ldots, X_n^{I_n})$ with LAIR [39] yields new domains $E_1, \ldots, E_n$, where

$$E_i = \{x_i \in I_i \mid \exists\, x_j \in I_j (j = 1, \ldots, n, j \neq i) \text{ s.t. } (x_1, \ldots, x_n) \in p^\epsilon\}.$$

**Theorem 4.1**: Let $I_1, \ldots, I_n$ be floating-point intervals and $p \in I\!R^n$. If $I'_1, \ldots, I'_n$ are output intervals of performing interval narrowing on $(p, \langle I_1, \ldots, I_n \rangle)$ and $E_1, \ldots, E_n$ are new domains obtained from applying LAIR to $p^\epsilon(X_1^{I_1}, \ldots, X_n^{I_n})$, then $I'_i = E_i$ for all $i = 1, \ldots, n$.

**Proof:** Let $J_i = \{x_i \mid (x_1, \ldots, x_n) \in ((I_1 \times \cdots \times I_n) \bigcap p)\}$.

$$\begin{aligned} I'_i &= \xi(J_i) \text{ by theorem 3.7} \\ &= \{x_i \mid \exists x'_i \in J_i \text{ s.t. } x_i \in \xi([x'_i, x'_i])\} \text{ by lemma 3.3} \\ &= \{x_i \mid (x'_1, \ldots, x'_n) \in ((I_1 \times \cdots \times I_n) \textstyle\bigcap p) \text{ s.t. } x_i \in \xi([x'_i, x'_i])\} \\ &= \{x_i \in I_i \mid \exists\, x_j \in I_j, j = 1, \ldots, n, j \neq i \text{ s.t. } (x_1, \ldots, x_n) \in p^\epsilon\} \text{ since } \xi([x'_j, x'_j]) \subseteq I_j \\ &= E_i \end{aligned}$$

■

There are two implications of theorem 4.1.

*First, interval narrowing is an instance of LAIR for interval domains. Thus interval arithmetic based on interval narrowing can be incorporated in CHIP without changing its semantics. Second, interval narrowing is the basis of an implementation of LAIR for interval domains.*

Programmers, however, should be aware that the arithmetic predicates are defined in terms of the approximations $p^\epsilon$ of the corresponding relations $p \subseteq I\!R^n$.

LAIR for finite domains can be expensive; its cost growing with the size of the domains in a constraint. However, the cost of the interval narrowing operation (LAIR for interval domains) for each arithmetic relation $p$ is constant, regardless of the size of the interval domains.

## 4.4 Constraint Relaxation

The other essential component of relational interval arithmetic is algorithm 1. We show that algorithm 1 is embedded in the proof procedure of CHIP.

Derivations in CHIP consist of SLDD-derivation steps interleaved with LAIR. In logic programming, the computation rule is usually crucial. Prolog always selects the leftmost atom in a goal. The same computation rule is not suitable for CHIP because the leftmost goal may be a stable constraint. As interval narrowing and, thus, LAIR are idempotent, such selection would produce no effective computation. CHIP uses a computation rule that tries to avoid selecting stable constraints.

CHIP maintains a goal by two sets $Y$ and $N$ of distinct atoms. The set $N$ contains all the atoms submitted to a lookahead declaration that, if selected, will not change under the application of LAIR. $Y$ is the set of all the other atoms. Initially, $Y$ is the set of all atoms in the original goal and $N$ is the empty set. We denote a goal by $\leftarrow \langle Y, N \rangle$; $Y$ is called the *$Y$-part* of the goal and $N$ is the *$N$-part*.

CHIP uses the lookahead-efficient computation rule in its proof procedure. A computation rule $R$ is *lookahead-efficient* if and only if:

1. an atom submitted to a lookahead declaration is selected by $R$ only when it is ground or lookahead-available;

2. $R$ selects only atoms in the $Y$-part of the goal; and

3. if one or more atoms submitted to a lookahead declaration in the $Y$-part are lookahead-available or ground, then $R$ selects one of them.

Intuitively, an atom $A$ is lookahead-available if it is in a suitable form to be reduced by LAIR. It requires that $A$ is lookahead-checkable. That is because LAIR needs to check the satisfiability of ground instances of $A$ using some form of resolution. There is no such need in the case of interval narrowing and we can drop the requirement. For $p^{\epsilon} \subseteq \mathbb{R}^n$, we declare

lookahead $p^{\epsilon}(d, \ldots, d)$.

Therefore, the atom $p^{\epsilon}(X_1^{I_1}, \ldots, X_n^{I_n})$ is always lookahead-available.

The proof procedure of CHIP [39] is as follows. Let $\leftarrow \langle Y, N\rangle$ be a goal and $G$ be the atom selected by a lookahead-efficient computation rule. Let $\theta$ be the substitution resulting from the resolution of $G$ by either a SLDD-derivation step or LAIR and $I$ the set of atoms introduced by the reduction of $G$. $I$ is the empty set if LAIR is used and the body of the input clause otherwise. Let $YI$ be $Y \cup I \setminus \{G\}$ and $NY$ the set of atoms $Q$ in $N$ such that $Q\theta \neq Q$. The new goal is

1. $\leftarrow \langle YI \cup NY, \{G\} \cup N \setminus NY\rangle\theta$ if $G$ is lookahead-available and $G\theta$ contains more than one domain variable;

2. $\leftarrow \langle YI \cup NY, N \setminus NY\rangle\theta$ otherwise;

where $\langle X, Y\rangle\theta$ is the application of $\theta$ to all the atoms of $X$ and $Y$.

**Theorem 4.2**: The proof procedure of CHIP, using a lookahead-efficient computation rule, executes algorithm 1.

**Proof:** We show the truth of the theorem by the following arguments:

1. In algorithm 1, the active list $A$ is contained in the $Y$-part of the goal and the passive list $P$ is the $N$-part.

2. LAIR, as proved in theorem 4.1, performs interval narrowing (line 5 of algorithm 1).

3. $NY$ is the set of atoms that become potentially unstable since they share variables, the domains of which are narrowed, with the current reduced atom. $I$ is the empty set since LAIR is used. The computation of the new $Y$-part $YI \cup NY = Y \setminus \{G\} \cup NY$ is equivalent to line 4 and lines 10 to 14 of algorithm 1. The computation of the new $N$-part is equivalent to in lines 10 to 14 and line 16.

4. A lookahead-efficient computation rule ensures that if there exist constraints in the $Y$-part, then they will be selected. This executes the outer loop (lines 3 to 17) of algorithm 1 completely before other atoms are selected.

■

## 4.5 The ICHIP Language

We show in the last two subsections how CHIP can be extended with relational interval arithmetic in such a way that its logical semantics is preserved. ICHIP is such an extended language. It provides relational arithmetic on integers, rational and floating-point numbers. We also consider relational interval arithmetic in CLP($\mathcal{R}$) [17]. The extended language ICLP($\mathcal{R}$) [20] can only handle constraints on floating-point numbers. The integration into Prolog III [6] is under investigation.

Domains in ICHIP can also be intervals. Reduction of goals with arithmetic primitive predicates are performed by interval narrowing. The proof procedure of ICHIP consists of SLDD-derivation steps interleaved with complete executions of algorithm 1.

Let $X^D$ be a domain variable with domain $D$. We express the constraint $(p, \langle I_1, \ldots, I_n \rangle)$ in ICHIP as

$$p^\epsilon(X_1^{I_1}, \ldots, X_n^{I_n}).$$

Constraints share intervals when they share domain variables. For example, to solve the positive root of $x(x-1)-6=0$ with initial interval $[1,100]$, we pose the query:

$$\leftarrow \texttt{add}^{\epsilon}(X_1^{(-\infty,+\infty)}, 1, X^{[1,100]}), \texttt{multiply}^{+\epsilon}(X^{[1,100]}, X_1^{(-\infty,+\infty)}, 6).$$

Partitioning of relations can also be expressed compactly in ICHIP. For example, the `multiply` relation can be defined by

$$\texttt{multiply}^{\epsilon}\texttt{(X,Y,Z) :- le}^{\epsilon}\texttt{(0,X),multiply}^{+\epsilon}\texttt{(X,Y,Z).}$$
$$\texttt{multiply}^{\epsilon}\texttt{(X,Y,Z) :- lt}^{\epsilon}\texttt{(X,0),multiply}^{-\epsilon}\texttt{(X,Y,Z).}$$

While Cleary's approach [5] and BNR Prolog [30, 31] deviate from standard logic programming semantics, ICHIP remains in the CHIP framework. Most importantly, ICHIP derivations are logical inferences and *numerical computation is deduction* in ICHIP. One advantage of ICHIP over other CLP systems, such as CLP($\mathcal{R}$) [17], is the ability to handle non-linear constraints, which are delayed in CLP($\mathcal{R}$).

## 4.6 Domain Splitting and Answer Interpretation

It is well-known that arc consistency techniques are "incomplete" [23]: a network can be stable but neither a solution nor inconsistency is found. Therefore, an ICHIP derivation can end with *floundered goals* [25], goals with atoms that cannot be selected by the computation rule. For example, this behaviour occurs if we use ICHIP to solve the roots of general polynomial and simultaneous equations [5]. In CHIP, floundering is avoided by the built-in predicate `indomain`, which implements a form of case analysis. The `indomain` predicate takes as argument a domain variable, and enumerates the elements in the domain of its argument. Intervals are infinite sets. Implementing a similar predicate is not feasible. For interval domains, we use another case analysis technique: *domain splitting*. Cleary [5] discusses two predicates, `linear_split` and `exp_split`, that split an interval into two partitions, visit one of them, and the other upon backtracking. `Linear_split` and `exp_split` split at different points of the interval. CHIP also provides a similar built-in predicate `split`, which handles finite domains. This can easily be extended to handle interval domains. We can use domain splitting to narrow the interval domains until they reach an acceptable width or the precision of the underlying floating-point system.

Domain splitting can be used to narrow intervals to a desirable width but it may not find solutions either. Floundering can still occur. We call the floundered goals *incomplete solutions*. Logically, incomplete solutions can be interpreted as qualified or conditional answers [40, 4]. 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 $\theta$ 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 [39], 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 [16].

# 5 Suggestions For Further Work

**Efficient Implementation.** We need an implementation of ICHIP to validate the feasibility of our framework. It is also important to conduct empirical study of the performance of the language. There is potential parallelism in algorithm 1. We expect Leung's doctoral dissertation [21] to be the basis of a parallel or distributed implementation of algorithm 1.

**Improving the relaxation algorithm.** The efficiency of a relational interval arithmetic system depends critically on algorithm 1. As described, our relaxation algorithm is similar to Mackworth's AC-3 and the Waltz algorithm. Techniques, such as those proposed in [27, 10], may be applicable to our relaxation algorithm to improve its efficiency. Another approach is to use heuristics to order interval constraints in the active list $A$. A useful heuristic is the first-fail principle [12], which states: *to succeed, try first where failure is most likely.*

**Complexity Analysis** The complexity of an interval narrowing operation is easy to determine since it depends only on the number of floating-point operations performed by each $F_i(p)$ function. Besides, it is independent of the size of the interval domains. It is difficult, however, to formulate a complexity measure for algorithm 1 since its termination depends on the precision of the underlying floating-point system. Such a measure must take the precision into account. A starting point for this work is to identify and study some simple classes of interval networks.

**Variable identity.** Suppose $X$ is the interval $[-1, 2]$. The result of the interval expression $X \otimes X$ is $[-2, 4]$. Since $X \times X$ in real arithmetic is the squaring operation, some readers may have anticipated the value of the interval expression to be in the positive region of the real line. The problem is that interval arithmetic simply multiplies $[-1, 2]$ by $[-1, 2]$ without realizing that the same variable $X$ is used as both the multiplier and the multiplicand, resulting in a larger than desirable interval. We call this the *variable identity* problem of interval arithmetic. To obtain $[1, 4]$ as the answer, we need to use the interval squaring operation $\mathrm{SQR}(X)$, where $\mathrm{SQR}(X) = \{x^2 \mid x \in X\}$.

This variable identity problem also occurs in interval narrowing, so that the output intervals are larger than expected. The culprit is multiple occurrences of a variable in an interval constraint. A solution is based on the following observation, using the `multiply` relation as an example:

$$\begin{aligned}
&\forall X, Y : (X, X, Y) \in \texttt{multiply} \Leftrightarrow (X, Y) \in \texttt{square}, \\
&\forall X, Y : (X, Y, X), (Y, X, X) \in \texttt{multiply} \Leftrightarrow (X, Y) \in \texttt{zeroOrOne}, \\
&\forall X : (X, X, X) \in \texttt{multiply} \Leftrightarrow (X) \in \texttt{zeroAndOne},
\end{aligned}$$

where $\texttt{square} = \{(x, y) \mid y = x^2\}$, $\texttt{zeroOrOne} = \{(x, y) \mid x = 0 \text{ or } y = 1\}$, and $\texttt{zeroAndOne} = \{(0), (1)\}$. When we encounter a `multiply` constraint at runtime, we select an equivalent relation using the instantiation pattern of the variables. For example, the constraint $(\langle X, X, Y\rangle, \texttt{multiply})$ becomes $(\langle X, Y\rangle, \texttt{square})$. This runtime optimization will help us to obtain smaller output intervals.

**Prolog III and CAL.** Prolog III and CAL refrain from floating-point arithmetic in order to ensure soundness of their numerical computations. It would be interesting to see if the two constraint logic programming languages can be extended with relational interval arithmetic.

# 6 Acknowledgements

Paul Strooper and Peter Walsh gave constructive comments to early drafts of this paper. We thank the anonymous referees for their insightful comments. We are grateful to Stott Parker for pointing out the variable identity problem. Generous support was provided by the British Columbia Advanced Systems Institute, the Canada Natural Science and Engineering Research Council, the Canadian Institute for Advanced Research, the Institute of Robotics and Intelligent Systems, and the Laboratory for Automation, Communication and Information Systems Research.

# References


[1] Alefeld, G. and Herzberger, J., *Introduction to Interval Computations.* Academic Press, New York, 1983. Translated by J. Rokne from *Elemente der Intervallrechnung.*

[2] Ashby, W.R., Constraint analysis of many-dimensional relations. *General Systems, Yearbook of the Society for General Systems Research*, 9:99–106, 1964.

[3] Bundy, A., A generalized interval package and its use for semantic checking. *ACM Transactions on Mathematical Systems*, 10(4):397–409, 1984.

[4] Cheng, M.H.M., van Emden, M.H., and Strooper, P.A., Complete sets of frontiers in logic-based program transformation. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 283–297. MIT Press, 1989. Also as Technical Report DCS-74-IR (LP-3), University of Victoria, Victoria, Canada, April, 1988.

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

[6] Colmerauer, A., An introduction to Prolog III. *Communications of the ACM*, pages 69–90, July 1990.

[7] Davis, E., Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[8] de Kleer, J. and Brown, J.S., A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[9] Dean, T., Temporal imagery: An approach to reasoning about time for planning and problem solving. Technical Report 433, Yale University, New Haven, CT, USA, 1985.

[10] Deville, Y. and Van Hentenryck, P., An efficient arc consistency algorithm for a class of CSP algorithm. In *Proceedings of IJCAI 1991*, pages 325–330, 1991.

[11] Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F., 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.

[12] Haralick, R.M. and Elliot, G.L., Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[13] Havens, W.S., Sidebottom, S., Sidebottom, G., Jones, J., Cuperman, M., and Davison, R., 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.

[14] Hayes, P.J., Computation and deduction. In *Proceedings of the Second MFCS Symposium*, pages 105–118. Czechoslovak Academy of Sciences, 1973.

[15] Hyvönen, E., Constraint reasoning based on interval arithmetic. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1193–1198, Detroit, USA, 1989.

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

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

[18] Kaushik, V., Erratum of logical arithmetic. Personal Communication, Department of Computer Science, University of Calgary, Alberta, Canada, 1990.

[19] Kirchner, R. and Kulisch, U., Arithmetic for vector processors. In R.E. Moore, editor, *Reliability in Computing—The Role of Interval Methods in Scientific Computing*, pages 3–42. Academic Press, 1988.

[20] Lee, J.H.M. and van Emden, M.H., 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. Also as Technical Report DCS-183-IR (LP-18), University of Victoria, Victoria, Canada, 1991.

[21] Leung, H.F., *Distributed Constraint Logic Programming.* PhD thesis, Imperial College of Science, Technology and Medicine, London, United Kingdom, 1992.

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

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

[24] Mackworth, A.K., Mulder, J.A., and Havens, W.S., Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.

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

[26] McDermott, D.V. and Davis, E., Planning routes through uncertain territory. *Artificial Intelligence*, 22:107–156, 1984.

[27] Mohr, R. and Henderson, T.C., Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

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

[29] Moore, R.E., editor. *Reliability in Computing—The Role of Interval Methods in Scientific Computing.* Academic Press, 1988.

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

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

[32] Older, W.J., The application of relational arithmetic to x-ray diffraction crystallography. Research Report 89001, Computing Research Laboratory, Bell-Northern Research, Ottawa, Ont., Canada, February 1989.

[33] Rump, S.M., Algorithms for verified inclusions—theory and practice. In R.E. Moore, editor, *Reliability in Computing—The Role of Interval Methods in Scientific Computing*, pages 109–126. Academic Press, 1988.

[34] Sidebottom, G. and Havens, W.S., Hierarchical arc consistency applied to numeric constraint processing in logic programming. Technical Report CSS-IS TR 91-06, Centre for Systems Science, Simon Fraser University, Burnaby, B.C., Canada, 1991.

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

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

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

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

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

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

[41] Waltz, D., Understanding line drawings of scenes with shadows. In P. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, 1975.

[42] Zweben, M. and Eskey, M., Constraint satisfaction with delayed evaluation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 875–880, Detroit, USA, 1989.