Constraint Arithmetic on Real Intervals

William Older
André Vellino

Abstract

Constraint interval arithmetic is a sublanguage of BNR Prolog which offers a new approach to the old problem of deriving numerical consequences from algebraic models. Since it is simultaneously a numerical computation technique and a proof technique, it bypasses the traditional dichotomy between (numeric) calculation and (symbolic) proofs. This interplay between proof and calculation can be used effectively to handle practical problems which neither can handle alone. The underlying semantic model is based on the properties of monotone contraction operators on a lattice, an algebraic setting in which fixed point semantics take an especially elegant form.

1 Introduction

Procedural arithmetic spoils the declarative nature of pure Prolog programs. Programs written in pure Prolog should be readable declaratively and the order of logical expressions (conjunctions, disjunctions and negations) should be semantically irrelevant. Moreover, they should preserve the following properties:

- narrowing (the answer is a substitution of the question)
- idempotence (executing the answer adds nothing) and
- monotonicity (the more specific the question is, the more specific the answer will be).

Relational arithmetic in constraint logic programming languages typically applies to either finite or integer domains (CHIP) [5,11] rationals (Prolog-III) [3] or floating point numbers (CLP($\Re$)) [7]. In these systems the set of constraints is effectively restricted to linear equations or inequalities, for which there are well-known solution algorithms.

Another approach suggested by John Cleary in [2] and also proposed independently by Hyvönen in [6], considers relational arithmetic on continuous domains by adapting techniques from interval arithmetic. These ideas for constraint interval

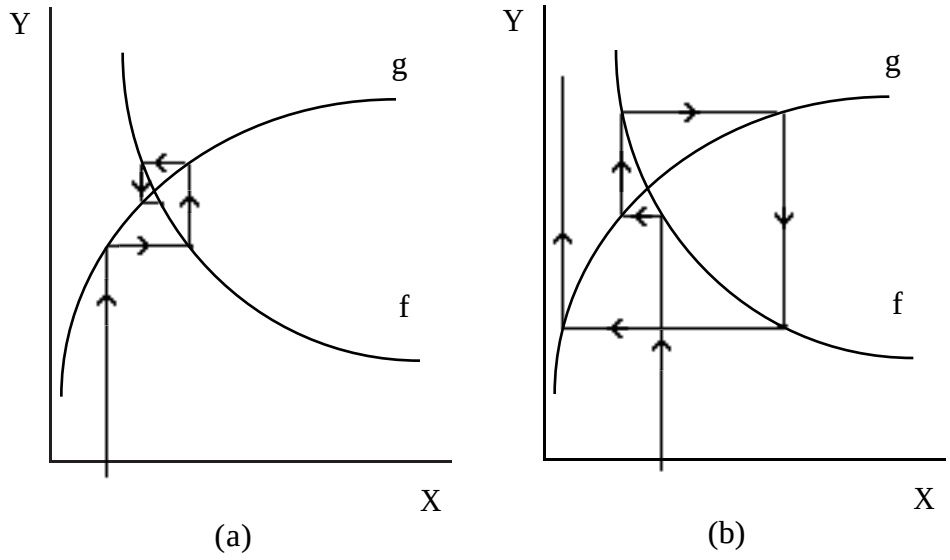

Figure 1: Conventional Fixed-Point Iteration

arithmetic were first fully developed and integrated into BNR Prolog in 1987 at Bell-Northern Research (BNR). We have also recently implemented constraint interval arithmetic in generic Prolog in order to have a portable version which could be executed independently of BNR Prolog.

Constraint interval arithmetic can be difficult to describe adequately because its conceptual framework is quite different from either conventional numerical analysis or algebraic and symbolic approaches. Section 2 of this paper describes the basic ideas in a direct, intuitive and operational fashion. Section 3 compares the constraint interval arithmetic with other approaches and discusses its practical ramifications. Section 4 approaches the subject from a rather different angle. The object here is to describe a lattice-theoretic framework in which to understand the properties of both the primitive interval operations and the constraint propagation networks created from them. This formalises some of the intuitions that are alluded to in the earlier sections but this material is self-contained and may be omitted on first reading.

2 Interval Iteration

One of the oldest and most widely used general methods for numerically solving non-linear equations is the basic fixed point iteration. Figure 1(a) shows how it is supposed to work in the simplest interesting case, that of finding a solution to the pair of equations $\{y = f(x), y = g(x)\}$ where f is decreasing and g increasing. Successive evaluations according to $y' = g(x)$, $x' = f^{-1}(y')$ are expected to converge to the (unique) solution of the two equations.

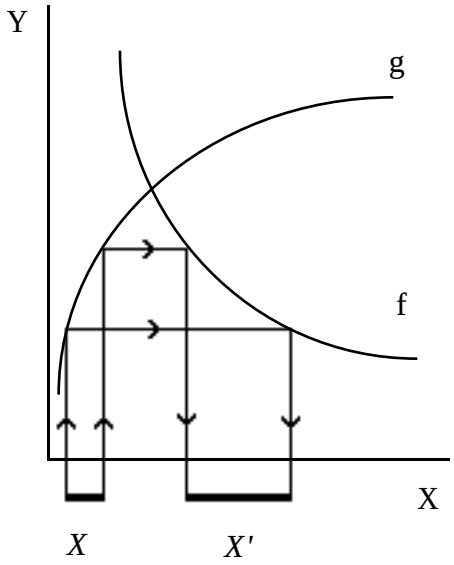

Figure 2: Interval Failure

Unfortunately, in practice, one often encounters behavior like that shown in Figure 1(b). After appearing to converge at first, the iterations of x wander away from the desired solution. Eventually the iterations may find another, unexpected, solution or may become periodic, or even chaotic. Whether the iteration converges or not depends on just about everything: the shape of the curves, the starting point, which function is applied first, the choice of coordinate system. Making the right choices for all these factors so as to guarantee convergence can require considerable analysis. Often in interactive contexts it is expedient to just try something at random, and if it appears to converge, accept the result. But the appearance of convergence (as this example shows) is not necessarily convergence, and there have been some cases of serious errors due to accepting such approximate solutions. Some more sophisticated techniques, such as the well-known Newton's method, consist of doing a symbolic transformation on the problem, and then iterating the transformed problem. The transformation serves to speed the convergence (if it converges), but since the iteration technique is the same it has the same qualitative difficulties.

If, however, the fixed point iteration is performed not on floating point values but on interval-valued variables, these difficulties vanish. An interval iteration is shown in Figure 2. Start with an interval X of values for x, and find the image interval $Y' = g(X)$, then the pre-image of that $X' = f^{-1}g(X)$ and then intersect it with the original interval. In this case the intersection is empty, and it follows that there was no solution in the original interval X.

It is important to see that this interval iteration is a proof that no solution exists and that it is justified by the fact that the functions f and g are monotonically

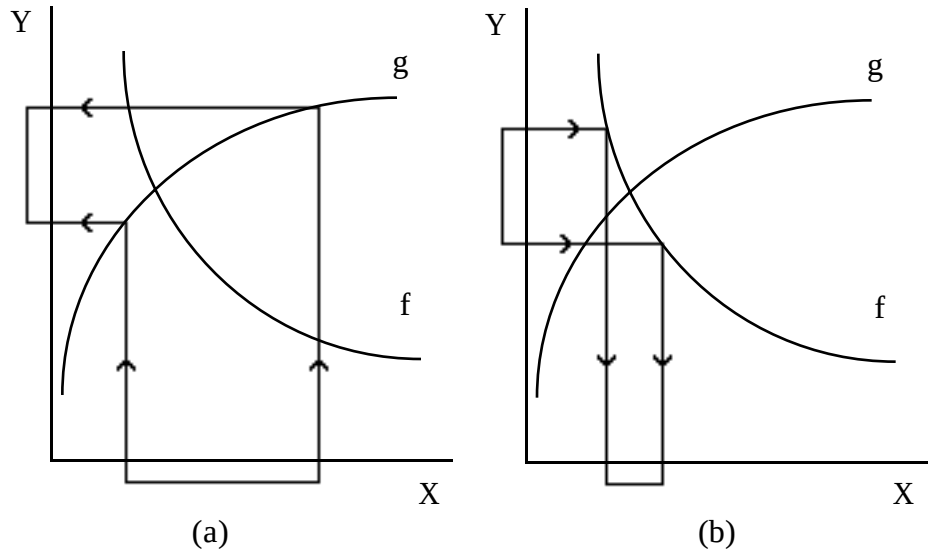

Figure 3: Interval Iteration

increasing or decreasing.

> **Proof**: Assume $x_L \leq x \leq x_H$ and x is a solution to $x = f^{-1}g(x)$. Then $g(x_L) \leq g(x) \leq g(x_H)$ since g is monotonically increasing and $x'_L = f^{-1}g(x_L) \geq f^{-1}g(x) \geq f^{-1}g(x_H) = x'_H$ since f is monotonically decreasing. Hence $x \in [x_L, x_H] \cap [x'_H, x'_L]$ but since the intersection is empty, the assumption that $x_L \leq x \leq x_H$ is a solution must be false.

For an initial interval that does contain a solution, Figure 3 shows the first two steps of a converging iteration in which the starting interval includes the solution and at each step the interval gets narrower. Ideally this iteration could be continued indefinitely and in the limit it converges to the solution. But practically speaking, few iterations often suffice to narrow the initial interval to a small final interval.

An argument similar to the above proof shows that *if there is a solution in the initial interval, then there is a solution in the final interval*. Of course, in this case, we might be inclined to believe that there is a solution because we believe in the completeness of the real numbers, but the constructive proof does not assume completeness thus cannot draw so definite a conclusion.

If this procedure is carried out using finite precision arithmetic, such as ordinary floating point, it does not generally work since rounding errors will often cause the intervals to miss each other. To avoid this problem with finite precision it is necessary to make each interval slightly bigger by "outward rounding" after every arithmetic operation. When this is done, the iteration stops automatically when it hits the limits imposed by the precision being used.

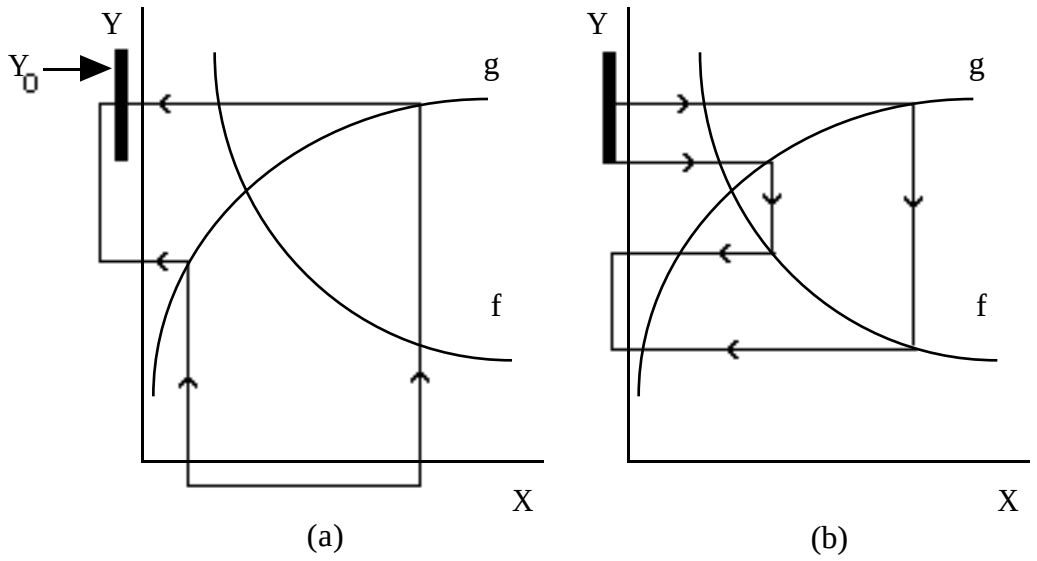

Figure 4: Inconsistent Initial Intervals

Figure 4 shows a more complex case leading to a contradiction. In this case both X and Y have initial intervals. There are a number of different ways in which one can apply the iteration. One can start with X and fold in the interval for Y after the first half-iteration (as indicated by the arrows in the figure), or start with Y and fold in the interval for X, or do both in parallel and merge the results. Whichever way one chooses, the result (in this case "failure") is the same. Even in non-failure cases the final intervals will have the same bounds regardless of the sequence chosen.

After studying some examples like this, one begins to suspect that, unlike traditional fixed point iterations, *there are no pathological cases*. The worst thing that can happen is that the initial interval does not narrow at all, which simply indicates that the original hypothesis is not strong enough to reach any useful conclusion.

3 Constraint Interval Arithmetic

In general terms, constraint interval arithmetic can be described as a method for taking a set of mathematical relations between certain quantities, and constructing from them a process which maps initial intervals on all quantities to final intervals for these quantities. It does this by constructing a proof (using interval fixed point iteration) that any solutions in the initial intervals must lie in the final intervals. The mapping from initial to final intervals is contracting (the final intervals are subintervals of their initial values), idempotent (repeating the same constraint has no effect on the interval values), and inclusion monotone (smaller initial intervals yield smaller final intervals). Moreover, the mapping from initial to final intervals is independent of the order in

which constraints are imposed and of the details of the implementation.

3.1 The Prolog Setting

Our implementation of constraint interval arithmetic is embedded in an experimental Prolog system, BNR Prolog, which serves as a meta-language for formulating mathematical models, posing the questions, and displaying the answers. Thus interval expressions form a distinct sublanguage of Prolog in this model, rather than being integrated at the level of unification, and this facilitates the use of Prolog as a meta-language. The language of model formulation includes all the basic operations on reals ($+$, $-$, $*$, $/$, exp, min, max, abs, sin, cos, etc.), the relations of equality ($==$), definition (is), and inequality ($\leq$), as well as some specialized "meta-predicates" for declaring intervals and querying the current size of intervals. The constraint system is fully incremental: any interval relation encountered during Prolog execution is automatically added to the constraint network.

To maintain backwards compatibility with conventional floating point arithmetic in Prolog, the convention is used that arithmetic statements encountered during a Prolog execution are treated as interval constraints if they refer to any interval object, and as conventional arithmetic otherwise. This allows conventional arithmetic programs to be used in "interval mode" by simply calling them with interval inputs/outputs.

The constraint interval language is fully relational, so that, for example, execution of a statement such as

```
R*R == X*X + Y*Y
```

(with `X`,`Y`,`R` intervals) can be used to calculate not only `R` from `X` and `Y`, but `X` from `R` and `Y`, and `Y` from `R` and `X` as well, depending on circumstances. The use of a relational arithmetic language, as with relational programming in Prolog, can greatly simplify the formulation of problems while at the same time making code much more general.

Interval arithmetic in BNR Prolog is integrated into the Prolog backtracking mechanism, so that a contradiction discovered in the constraint network automatically triggers backtracking to the last Prolog choice point, and "undoes" any subsequent changes to the constraint system (interval narrowings or addition of constraints). Consider, for example, the program that defines the intersection of the folium of Descartes and an exponential decay.

```
folium_exp(X,Y) :-
                range(X,_),
                range(Y,_),
                X*X/Y + Y*Y/X == 2,
                Y == exp(-X).
```

The question `?- folium_exp(X,Y)` produces the answer

```
X = [-44.362, 1.8440e+19],
Y = [0.0000, 1.8440e+19]
```

which is insufficiently precise to be useful. However, the answer to the question `?- folium_exp(X,Y),(X >= 0.5 ; X =< 0.5)` produces two point solutions:

```
Y = [0.41961, 0.41962],
X = [0.86841, 0.86842] ;

Y = [0.74485, 0.74486],
X = [0.29456, 0.29457]
```

Thus the integration of interval constraints with backtracking is particularly useful for writing numerical search algorithms, or dealing with problems that have both combinatorial and continuous aspects. A more complete description of the surface syntax and operational appearance of constraint interval arithmetic can be found in [10], which also includes examples illustrating various features and [1] which compares BNR Prolog with other CLP systems.

3.2 Comparisons

It is helpful to contrast constraint interval arithmetic both with other constraint satisfaction techniques and with conventional methods. In the following subsections we note the differences from symbolic methods, procedural interval arithmetic and conventional error analysis methods. Moreover we emphasise the fact that constraint interval computations are *proofs*, which cannot be said of traditional numeric techniques.

3.2.1 Symbolic methods

It is crucial to note that constraint interval arithmetic is fundamentally a numeric and not a symbolic technique. The "program" is defined by mathematical relations, but questions are always numeric (the initial intervals) and solutions are always numeric (final intervals). This is quite unlike (for example) the behaviour of CLP($\Re$) on non-linear problems, where the questions are the relations themselves, and the answers will be a reduced set of symbolic relations.

Symbolic methods have, of course, the great advantage of generality, when they can be applied. But a numeric approach is often necessary because there is no symbolic solution, such as roots of polynomials of degree greater than 4, or because the symbolic solution is too unwieldy, as with linear systems. Furthermore, numeric methods can take advantage of the specific quantitative situation in ways that are not available to general symbolic techniques. For example, constraint relations which

are topologically close (in the sense of having "nearby" graphs) will generally behave in a qualitatively similar fashion, although one may be symbolically tractable and the other symbolically intractable. In particular, the difference between a linearized model and a "nearby" mildly non-linear refinement (a commonly occurring case in engineering) does not significantly change the nature of the problem from the perspective of constraint interval arithmetic; indeed, experience suggests that the non-linear problem will often work better.

Even though interval arithmetic is a numeric technique it enhances any symbolic methods with which it is made to cooperate. Unlike conventional floating point, interval arithmetic formally honors the algebraic axioms of real arithmetic, in the sense that symbolic transformations based on those axioms can never lead to bogus contradictions due to rounding problems. Thus "redundant" relations, which conventionally can cause problems because of such bogus contradictions, become a positive enhancement instead of a problem. For example, in a linear system problem in constraint interval arithmetic, a pivoting operation produces an additional redundant equation rather than a replacement for one of the original relations.

The numeric nature of this technique implies that any system of finite precision floating point arithmetic with which it is implemented renders the precise bounds of computed intervals sensitive to the specific formulation of the problem. For example, Y_1 `is` $(A+B)+(C+D)$ and Y_2 `is` $(((A+B)+C)+D)$ may result in different computed bounds for Y because of the non-associativity of floating point arithmetic. However, since both Y_1 and Y_2 are supersets of the answer Y, $\emptyset \neq Y \subseteq Y_1 \cap Y_2$ is always true. Given a particular one of the many mathematically equivalent formulations (i.e. a specific set of parentheses for an expression) the theory developed in section 4 then implies that final bounds are independent of the order in which the constraints are processed.

3.2.2 Functional interval arithmetic

Constraint interval arithmetic obviously has much in common with the classical interval arithmetic developed by Moore in the 1960's, and much of the classical interval literature can be very useful [8,9]. However, as a whole, they are extremely different paradigms.

The major difference is that constraint interval arithmetic is a *relational* language while the classical interval arithmetic is *functional*. This difference affects not only the formal structure of the language, but has a major impact on problem formulation. The second difference is the notion of a constraint as something which can be "imposed" once, but which continues to "operate" so as to maintain its truth; classical interval arithmetic merely provides an abstract data type for intervals, and each individual calculation must be explicitly coded. Finally, the Prolog metalanguage with integrated backtracking and the logic programming paradigm radically changes the way in which one formulates problems.

3.2.3 Error analysis

Since interval based techniques effectively perform an error analysis along with each computation, numerically fragile formulations and even conditionally fragile behaviour for specific inputs show up immediately as unexpectedly wide intervals which refuse to "narrow" even when inputs are narrowed. This provides a useful indicator of possible problems with a model (or of the system being modeled) for which there is no simple conventional equivalent.

For many numerical applications much of the work lies in determining that the model is sufficiently robust for small variations in many parameters. Traditional methods of sensitivity analysis based on error-propagation formulae, calculus or just repeated evaluations, are laborious, error-prone, and often inconclusive because they cannot provide the desired guarantees. A wide range of sensitivity/stability analyses can be done very easily using interval arithmetic without requiring any additional code. Furthermore, unlike traditional methods, such interval calculations can guarantee (modulo the correctness of the model) that if some parameters (e.g. component specifications) are within certain ranges, then some other variables (e.g. overall system parameters) will be within certain ranges under worst-case conditions.

Since interval fixed point iterations stop automatically when they reach the resolution limit of the underlying arithmetic, there is no need to use specific convergence tests which are such a common and often problematic feature of conventional numerical programming.

3.3 Computations as Proofs

The other aspect of constraint interval arithmetic, and one which makes it very different from traditional numerical techniques, is that its computations represent proofs, and these are always proofs of the non-existence of solutions. As proofs, they carry a degree of logical force generally absent from traditional floating point numerical computing, which is, by comparison, concerned only with heuristics. The fact that its proofs are always non-existence proofs also makes it very different from traditional exact (rational) arithmetic, in which computations can be thought of as constructive proofs of existence.

It is precisely because interval proofs are non-existence proofs that they can be interpreted as referring to the mathematical reals, even though only bounded precision constants actually appear in the proofs themselves and the constraint system itself has no notion of "real number" in the full mathematical sense. (Of course, these proofs refer as well to the rationals, computable reals, and non-standard reals.)

To take full advantage of this proof aspect of the technique, it is sometimes necessary to formulate problems negatively, so that a "failure" indicates a successful proof. Thus in the problem of formal system verification mentioned above, one asks questions of the form: if components all lie within their specified tolerance intervals, can

a system parameter lie outside its specification? A "no" answer then indicates that a successful proof of compliance has been constructed, thus achieving a formal verification for all systems characterized by the model and initial intervals. If, however, a contradiction is not found, then the final intervals indicate conditions in which the specifications might not be met, and thus provide a direct indication of where the design may be marginal.

The art of computation-as-proof is particularly striking in interval search techniques, as in problems of global non-linear optimization, where it is used to eliminate subregions where the optimum cannot occur. These algorithms, essentially of the branch-and-bound type, were originally developed in the context of functional interval arithmetic, but become much more elegant in the context provided by Prolog and relational interval arithmetic where Prolog backtracking automatically manages the housekeeping chores associated with branching, while the interval machinery provides the necessary bounds automatically.

3.4 Implementation Considerations

Individual interval primitives such as addition (subtraction) and multiplication (division) can be implemented in a manner which requires on average less than three floating point operations per execution, not including comparisons. However, since a considerable number of comparison operations are involved in each operation, it is important to use a floating point representation such as the IEEE standard which supports cheap comparisons.

The complete insensitivity of the result of the computation to the order in which the individual primitive operations are done would seem to make constraint interval arithmetic an excellent candidate for low-level parallel implementations. The computation is so confluent that it is possible to take a totally relaxed attitude about synchronization of everything except failure and the subsequent backtracking.

Comparison of memory costs between constraint interval arithmetic and alternative systems is difficult. With regard to static costs, the mathematical model is represented in constraint systems by a data structure while conventionally it may be largely represented by an algorithm and counted as code. It should be noted that since a constraint corresponds to each execution instance of an arithmetic expression, the size of the constraint system is not related in any simple way to the size of the arithmetic part of the program which creates it, and in fact the largest problem we have dealt with so far has a program of only four pages in which only about six lines even mention arithmetic.

Memory management overheads—chiefly the overheads of "trailing" changes so that states can be restored on backtracking–naturally seem to be higher than in conventional numeric contexts until one considers that these effectively replace initialization code. These overheads are much smaller than the usual implementations of floating point arithmetic in Prolog, which tend to produce substantial amounts of

garbage which may require collection.

3.5 Complexity Issues

For constraint networks that are acyclic it is possible to bound the number of operations required to achieve a fixed point, and the bound is linear in the size of the network. To restore a fixed point after a change requires much less work, and, in particular cases, may require only a few operations.

There is a somewhat wider class of constraint networks that are essentially functional in that if the individual constraints were to be executed functionally in some particular order, they would produce the required fixed point.

Empirically it has been observed that such networks seem to evaluate in linear (or near-linear) time, but this probably depends on the specific details of the internal scheduling mechanism being used to guide propagation and is not a general property of the technique. One example of this kind is linear systems which are (both row and column) permutations of a triangular system, with some interesting special cases in polynomial multiplication, reversion of series, and matrix factorization. A more subtle example is with PERT/CPM scheduling problems, where the algorithm seems to "find" its way to the classical canonical decomposition into separate upper and lower-bound potential construction problems regardless of the order in which the problem is stated. This ability to find and exploit hidden functional pathways (which seem to be quite common in industrial problems) may be one of the main explanations for the otherwise puzzling fact that this technique has worked better than expected on realistic problems. It should be noted, also, that whether such functional pathways exist and which way the data flow goes depends on the relative sizes of intervals, and the optimal order of evaluation may be quite different for different initial conditions.

For general constraint networks containing loops, it is very difficult to predict performance theoretically, even for specific problems, just as it is generally difficult to predict the number of iterations to convergence in conventional fixed point iterations. It is also unclear how to formulate a useful complexity measure for constraint interval arithmetic in general. Since termination is governed by the actual precision being used, a complexity model must take the precision into account. Worst-case performance for fixed precision in general has an upper-bound that depends exponentially on the number of variables but is independent of the problem being solved. This bound, which is on the order of the number of different states of the system, is so large as to be practically useless, and there are some relatively simple problems that come close to achieving it. For example:

```
abs(X) =<  M, abs(Y) =< M,
A*X + B*Y == C
A*X + B*Y == C'
```

where `C` and `C'` are disjoint intervals which differ by some small δ. This eventually

results in failure, but effectively does so by counting from -M to +M by steps of size δ, which can be an enormously large number. On the other hand, it has been our experience that, in practice, quite respectable industrial-sized problems with hundreds of variables and constraints, will, in most cases, converge to a fixed point in a reasonable amount of time. In particular, if the constraints are inconsistent, failure usually occurs fairly quickly unless the initial conditions are very near the boundary of the failure region.

4 Theory

The general theory of constraint interval arithmetic falls naturally into three distinct layers. The top layer is concerned with the conversion from the external source language to an internal data structure or constraint network. The bottom layer is a simple abstract theory for the implementation of primitives. The middle layer concerns the interval iteration and relates the properties of the primitive operations to that of the constraint network as a whole. It is to this middle layer (subsection 4.3) that this section is mainly devoted, since its theory is rich.

4.1 Compilation

The compilation step that transforms an arithmetic expression into a set of nodes in the constraint network is much like a conventional compilation process in that input expressions are broken down into atomic operations corresponding to the primitives, which are then mapped directly. As in conventional compilation, this can be done either very simply (as in the current BNR Prolog implementation) or in complex ways designed to optimize performance. In the BNR Prolog implementation, for example the expression

```
Y * (Y+1) == X * (Y - 1)
```

(with X, Y intervals) is transformed internally into the conjunction of primitive interval constraint relations

```
add( Y, 1, Y1),
multiply(Y, Y1, Y2),
add(Y3, 1, Y),
multiply(X, Y3, X1),
equal(Y2, X1)
```

(where Y1, Y2, Y3, X1 are new interval terms).

Many of the optimizing principles of standard compilation techniques, such as factoring of common subexpressions, can be applied here too. The major difference from conventional numerical compilation lies in the fact that such mathematically justified transformations cannot lead to incorrect behavior due to round-off problems.

4.2 Primitives

The theory behind the primitive relations (`add`, `multiply`, etc.) is straight-forward. Given a relation R of arity n over the reals, and n intervals $X(j)$ with representable bounds, intersect R with the Cartesian product of the $X(j)$, project the result back onto each coordinate axis and then round outward to the smallest containing interval whose bounds are representable in the underlying machine representation. It is evident that this procedure is safe (never losing a possible solution), and that each output interval is smaller (in the sense of set inclusion) than it was originally. The result for each argument is an inclusion monotone set function of all its inputs since it is a composition of inverse projection maps, intersection, projections, and a closure operation, each of which is itself inclusion monotone. Finally, the operation can also be shown to be idempotent, essentially because everything that can be trimmed gets trimmed the first time.

4.3 Fixed Point Semantics

The major theoretical component concerns the interval iteration process and relates the properties of primitive operations to that of the constraint network as a whole. A general theory, which must apply to all constraint networks formed from such primitives, must be based only on the common properties of the primitives and is necessarily rather abstract. The key question for such a general theory is to characterize the nature of the interval iteration process and its fixed points, relate these to initial conditions, and show how these properties are affected by the addition of new constraints. Since new constraints are merged by the same method that primitive operations are handled within a single constraint, a single analysis covers both situations.

4.3.1 States

For a fixed constraint network, the only things that change during the iteration process are the bounds of intervals: these are the natural states of the network. Since the intersection of two intervals is an interval (or failure) and since intervals can be partially ordered by set inclusion, and given that these properties can be extended to network states, it is useful to treat states as a meet semi-lattice $\mathcal{L}$ with the partial order denoted by $\preceq$ and meet (intersection) denoted by $\wedge$ (for an introduction to lattice theory see [4]). A bottom element $\perp$ (representing Prolog failure) is also added so that meet is always defined. A top element $\top$ represents the largest possible state, e.g. with all intervals ranging from $-\infty$ to $+\infty$.

Most states thus correspond to "boxes" in R^n, that is closures of the basic open neighbourhoods in the box topology in R^n. We assume that states are also closed under arbitrary meets, i.e. the semi-lattice of states is meet-complete. Of course, if finite-precision arithmetic is used the number of states is finite and completeness

holds trivially, but the more general theory has the advantage that it can be applied also to the infinite-precision case.

Since the semi-lattice has a top element and is assumed to be meet-complete, a standard construction allows us to define a join operation, $\vee$, by saying that the join of a collection is the meet of all elements larger than each item of the collection, and with this operation $\mathcal{L}$ becomes a complete lattice.

One theoretical advantage in using only closed intervals, and hence topologically closed states, is that if the top state is required to be compact then the states inherit the finite intersection property: if an arbitrary meet of states is $\perp$, a finite subcollection of those states meets at $\perp$.

4.3.2 Operators

The transition operators on states are represented as maps $p : \mathcal{L} \mapsto \mathcal{L}$ such that the product of two maps is their composition, which is therefore associative. The identity map is denoted by 1, and the constant map to $\perp$ is denoted 0. The lattice structure (and partial order) of $\mathcal{L}$ can be lifted into a lattice structure on operators by the usual pointwise definitions:

for all states A:

$$\begin{array}{rcl} p \preceq q & \iff & p(A) \preceq q(A) \\ (p \wedge q)(A) & = & p(A) \wedge q(A) \\ (p \vee q)(A) & = & p(A) \vee q(A) \end{array}$$

However, we are only interested in those maps which can be formed by composition of constraint interval primitives, and, as we noted above, these primitives are monotone and contracting:

N1. (contracting): $p(A) \preceq A$

N2. (monotone): $A \preceq B \Longrightarrow p(A) \preceq p(B)$

Since the product of contractions is a contraction and the product of monotones is monotone, we restrict our attention to the semi-group of monotone contractions on $\mathcal{L}$, denoted $MC(\mathcal{L})$.

$MC(\mathcal{L})$ contains 1, so it is a monoid, and 0 is an algebraic zero. $MC(\mathcal{L})$ is also partially ordered in the induced order and closed under the induced lattice operations, and 0 is its least and 1 its largest element. As a direct consequence of N1 and N2, the product operation is monotone in both variables: $p \preceq p'$ and $q \preceq q'$ implies $pq \preceq p'q'$. By definition,

$$\begin{array}{rl} (p \vee q)r(X) & = p(r(X)) \vee q(r(X)) \\ & = pr(X) \vee qr(X) \\ & = (pr \vee qr)(X) \end{array}$$

so right (but not left) multiplication distributes over join.

A similar argument shows that right multiplication distributes over meet. The monotonicity of product provides a substantial amount of right/left symmetry to the theory, but this symmetry is broken by the lack of left distribution laws.

4.3.3 Idempotents

In addition to satisfying N1 and N2, primitives are idempotent:

N3. (idempotent): $p(p(A)) = p(A)$

We will call such operators (those satisfying N1-N3) narrowing operators on $\mathcal{L}$, and denote the set of all narrowing operators by $N(\mathcal{L})$. $N(\mathcal{L})$ contains 0 and 1 and is partially ordered by $\preceq$, and we will be mainly concerned with its closure properties with respect to product and the lattice operations.

In any semi-group there are two relations that can be defined which capture the relative strength of idempotents:

$$\begin{aligned} pLq &\iff pq = p \\ pRq &\iff qp = p \end{aligned}$$

Clearly L is reflexive; it is transitive since $pq = p$ and $qr = q$ implies $pr = (pq)r = p(qr) = pq = p$, and similarly for R. In general, there may be no connection between the two relations, nor need they be partial orders. In $N(\mathcal{L})$, however, we have that $pq \preceq q$ (since p contracts) so $pq = p$ implies $p \preceq q$ and conversely if p is an idempotent, $p \preceq q$ implies $p = pp \preceq pq$. So $pq = p$ and hence $pLq \iff p \preceq q$. (Note that only p needs to be idempotent.) A similar proof applies to R, so both the left and right absorption orders are equivalent to $\preceq$.

If p and q are contractions, then since $p(X) \preceq X$ and $q(X) \preceq X$, we have by the least upper-bound property that $p(X) \preceq p(X) \vee q(X) \preceq X$, and if p is also monotone then $p(X) \preceq p(p(X) \vee q(X)) \preceq p(X)$ so $p(p(X) \vee q(X)) = p(X)$ and similarly for q. If p and q are both also idempotent, then

$$\begin{aligned} (p \vee q)(p \vee q)(X) &= (p \vee q)(p(X) \vee q(X)) \\ &= p(p(X) \vee q(X)) \vee q(p(X) \vee q(X)) \\ &= p(X) \vee q(X) \\ &= (p \vee q)(X) \end{aligned}$$

so $(p \vee q)$ is also idempotent.

4.3.4 Commutativity

Two primitive operations p and q which do not directly share any variables commute, $pq = qp$, so commutativity is very common in practice. If p and q are commutative idempotents, then pq is an idempotent since

$$(pq)(pq) = p(qp)q = p(pq)q = (pp)(qq) = pq$$

A fully commutative consistent network has an interesting structure related to Prolog. Since there are a finite number n of generating primitives, all of which commute, there are at most 2^n distinct products (including 1, but 0 is not in the generated set because we're assuming consistency). Some of these may be the same, however, as consequences of the order relation which is completely captured in the multiplicative structure by the absorptive relations. The absorption relation $yx = xy = x$ (where x and y are products of primitives) can be decomposed into those of the form $px = xp = x$ where p is primitive, since $x \preceq pq$ if and only if both $x \preceq p$ and $x \preceq q$. These can then be expressed in the form of Horn clauses $(p \leftarrow x)$, or, equivalently, as a set of functional dependencies. Thus the monoid structure of a consistent finite commutative constraint network is given by a finite propositional Horn logic.

4.3.5 Iteration

We need to iterate non-idempotent operators in order to create idempotent ones. Intuitively we would like to define iteration by

$$p^* \stackrel{def}{=} \lim_{n \to \infty} (p)^n$$

Since the sequence of powers of p is decreasing, the same effect is achieved by the definition

$$p^* \stackrel{def}{=} \bigwedge_{n=1}^{\infty} (p)^n$$

which exists because $\mathcal{L}$ is complete. We can then define the iterated product of p and q by

$$p * q = (pq)^*$$

Note that if p and q commute, then $p * q = pq$.

The operator determined by a constraint network can now be formally expressed as the * product of all its primitives, so every constraint network satisfies N1-N3. The loose statement that the result of interval iteration is "independent of the ordering" of operations can now be expressed (in part) by the assertion that $*$ is commutative and associative. Although these proofs can be carried out directly in $MC(\mathcal{L})$, a much more elegant proof proceeds indirectly via fixed points.

4.3.6 Fixed Points

A fixed point of a narrowing operator p is an element x of $\mathcal{L}$ such that $p(x) = x$. We will use the relation $fp(p, x)$ to say that x is a fixed point of p. The relation fp induces a Galois connection between sets of fixed points and operators which we will now investigate. For each narrowing operator p we define its family of fixed points by $F(p) = \{x : fp(p, x)\}$.

> **Lemma**: For p in $N(\mathcal{L})$, $F(p)$ satisfies:
>
> A1. $\bot \in F(p)$
>
> A2. $A_i \in F(p) \Longrightarrow \bigvee_i A_i \in F(p)$
>
> **Proof**: A1 is immediate. For A2 we note that since $A_i \preceq \bigvee_i A_i \Longrightarrow p(A_i) \preceq p(\bigvee_i A_i)$. So, from the definition of join, $\bigvee_i p(A_i) \preceq p(\bigvee_i A_i)$, but since all A_i are fixed points of p, $\bigvee_i A_i \preceq p(\bigvee_i A_i)$, and $p(\bigvee_i A_i) \preceq \bigvee_i A_i$ since p is contracting; it follows that $p(\bigvee_i A_i) = \bigvee_i A_i$.

Note that collections satisfying A1 and A2 are closed under arbitrary intersections in the power set of $\mathcal{L}$. Given a collection F in $\mathcal{L}$ satisfying A1 and A2, define an operator by $G(F)(x) = \{$ largest $a \in F : a \preceq x\}$. This makes sense because of A2, and it is easily seen that $G(F)$ is top preserving, contracting, monotone, and idempotent, so it is a narrowing operator. It is easily verified that these constructions define a bijective correspondence: $G(F(p)) = p$ and $F(G(F)) = F$.

The fixed points of $p * q$ are precisely the common fixed points of p and q. For if x is a common fixed point of p and q, then $pq(x) = x$ so $p * q(x) = x$, and conversely if $p * q(x) = x$ then $pq(x) = x$ (since $x = p * q(x) \preceq pq(x) \preceq x$), so $p(x) = ppq(x) = pq(x) = x$ and also $x = pq(x) \preceq q(x) \preceq x$ so $q(x) = x$. Then since $p * q$ corresponds to the common fixed points of p and q, we have

$$p * q = G(F(p) \cap F(q))$$

so $p * q$ is idempotent and $N(\mathcal{L})$ is closed under $*$. In particular the operator corresponding to a constraint network is idempotent and satisfies N1-N3. Furthermore $(N(\mathcal{L}), *)$ is isomorphic to the meet semi-lattice of sets (in $\mathcal{L}$) which satisfy A1-A2. In particular, $*$ is commutative and associative. However, this representation for $p * q$ also implies that any (composition) product of primitives which achieves a common fixed point when applied to any state x, will have computed the iterated product evaluated at x, and this confluence property provides a much stronger form of order independence and this what is actually used in practice.

Since the set of collections satisfying A1-A2 is closed under intersection and has a top element (the set of all elements in $\mathcal{L}$), it is possible to define a new join operation in the usual way, and transfer it to $N(\mathcal{L})$ via G to make $N(\mathcal{L})$ a complete lattice.

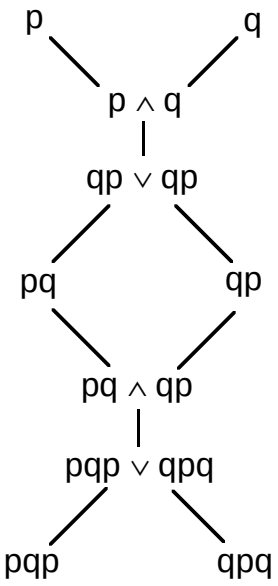

Figure 5: Lattice of Idempotents

This join operation is operationally the same as $\vee$, since $p \vee q$ is the least upper-bound in $MC(\mathcal{L})$ of p and q, and since it is idempotent it must therefore be the least upper-bound in $N(\mathcal{L})$ as well.

As an application of confluence, Figure 5 shows part of the lattice generated from two idempotents p and q by applying monotonicity and lattice properties. Confluence implies that there is exactly one idempotent in the diagram below $p \wedge q$, from which we can infer that (i) $(p \wedge q)^* = (pq \vee qp)^* = p * q$; (ii) $p \wedge q$ is idempotent $\Rightarrow p \wedge q = pq = qp$ is idempotent; (iii) $(p \wedge q)(p \wedge q) = pq \wedge qp$ is idempotent $\Rightarrow pqp = qpq$ is idempotent; (iv) pq is idempotent $\Rightarrow pq \wedge qp = pq = pqp$ is idempotent; (v) $(pq)^n = (qp)^n \Rightarrow$ both are idempotent, etc.

4.3.7 Compactness

If the underlying lattice is meet compact, then we get a finite failure principle. For if some finite constraint system ultimately fails on state x, then confluence implies that the iteration of the simple product p of all the primitives in the system fails, $(p*)(x) = \bot$. But by the definition of iteration, this is equivalent to a meet of all $p^n(x)$ failing, and meet compactness then implies that $p^n(x) = \bot$ for some finite n.

4.3.8 Constraint Graphs

The theory presented so far has been quite abstract and therefore somewhat remote from actual implementations. Something much closer to an implementation viewpoint can be constructed by defining an abstract graph or network associated with a definite

constraint system, given as a set of primitives. The nodes of this graph are the primitive operators; there is a undirected edge from one node to another whenever they do *not* commute. Figure 6 shows a such a graph for a model of paramagnetism in which $Y = a * X$, $Y = \tanh(X)$ and where boxes represent primitives and edges connect non-commuting primitives.

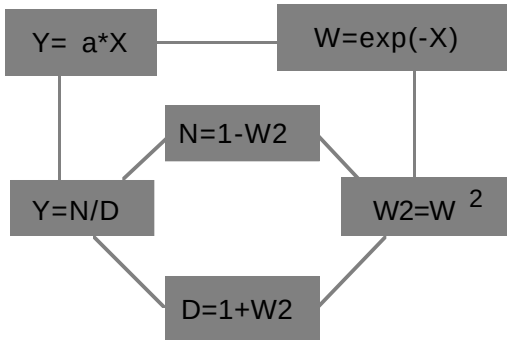

Figure 6: Constraint Network

Since each element of a connected component of the graph commutes with everything not in that component, the iterated product for the total network is just the commutative product of the iterated products over each separate component. Furthermore, when a node in a connected graph is evaluated, only its immediate neighbors may need to be reevaluated, so the graph serves to direct the reevaluation strategy. This network representation of a system of constraints plays a pivotal role in relating the abstract theory to concrete algorithms and specific problems.

4.3.9 Relation to Prolog

Constraint interval arithmetic is in some sense analogous to Prolog, or more precisely the non-backtracking part of BNR Prolog. In this analogy one takes $\mathcal{L}$ to be the space of terms with $\wedge$ corresponding to unification and $\perp$ to failure, and takes the state of a computation (along each separate branch of the non-deterministic computation) to be represented by a term (roughly the "goalstack").

Each call instance, *including external primitives*, is then conceived of as a narrowing operator on the state. Axiom N1 then holds because variables are only bound and never unbound. Axiom N2 eliminates "impure" constructs (such as instantiation tests and traditional arithmetic). Axiom N3 must be be replaced by

P3. (persistence) $A \preceq p(B) \Longrightarrow p(A) = A$

which captures the "write once" property of logic variables (and also eliminates primitives like `var/1`). P3 clearly implies N3. The set of operators satisfying {N1, N2, P3} will be denoted by $P(\mathcal{L})$.

The abstract theory of $P(\mathcal{L})$ is very simple, consisting of a single theorem:

Universal Answer Theorem: $p \in P(\mathcal{L}) \Longrightarrow p(X) = X \wedge p(\top)$.

Proof: $p(X) \preceq X$ by N1 and $p(X) \preceq p(\top)$ by N2, so $p(X) \preceq X \wedge p(\top)$. But from $p(X) \preceq X \wedge p(\top) \preceq X$ we get $p(X) \preceq p(X \wedge p(\top)) \preceq p(X)$ by N2 so $p(X) = p(X \wedge p(\top))$. Since $X \wedge p(\top) \preceq p(\top)$, P3 gives $p(X \wedge p(\top)) = X \wedge p(\top)$.

Thus persistence is a very strong axiom when combined with N1 and N2. However, N1 and P3 without N2 are satisfied by conventional Prolog arithmetic, which suffers from the problems which constraints are designed to eliminate.

5 Conclusion

The principal application domains for constraint interval arithmetic (of which we are aware) have been in the following areas: scheduling, interval temporal logic, protocol performance modeling, industrial design, parametric estimation in crystallography, and bounds analysis. It is significant that the interval approach to many of these problems is not comparable to conventional solutions, because either there are no known algorithms or conventional approaches are too difficult to apply in practice.

It is important to recognize that the basic technique of constraint interval arithmetic is relatively weak. This weakness reflects both its generality—which allows it to be applied usefully in situations in which no particular conventional algorithm is suitable—and the locality and simplicity of its primitives—which permit efficient implementations. For the same reason, however, there are relatively few problems for which it will be sufficient as a solution technique, just as there are relatively few problems in logic programming for which unification *alone* is a sufficient technique. The true power of the concept lies not so much in what it can accomplish in isolation but the way in which it combines synergistically with complementary techniques such as symbolic transformations and backtracking.

References

[1] Brown, R. G., Chinneck, J.W. and Karam, G. "Optimization with Constraint Programming Systems" in *Impact of Recent Computer Advances on Operations Research*, North Holland, January 1989.

[2] Cleary, J. C. "Logical Arithmetic", Future Computing Systems, **2** (2), pages 125–149, 1987.

[3] Colmerauer, A. "An introduction to Prolog III", *Communications of the ACM* **33**(7):69, 1990.

[4] Davey B. A. and Priestley H. A., *Introduction to Lattices and Order*, Cambridge University Press, Cambridge, 1990.

[5] Dincbas M., Simonis H. and van Hentenryck P. "Solving Large Combinatorial Problems in Logic Programming", *Journal of Logic Programming* **8**, (1&2), pages 72-93, 1990.

[6] Hyvönen, E. (1989) "Constraint Reasoning Based on Interval Arithmetic", Proceedings of *IJCAI 1989*, pages 193–199.

[7] Jaffar, J. and Michaylov, S. "Methodology and Implementation of a CLP system", *Proc. 4th Int. Conf. on Logic Programming*, J-L. Lassez (Ed.), MIT Press, 1987.

[8] Moore, R. E. (Ed.) *Interval Analysis*, Prentice Hall, New Jersey, 1966.

[9] Nickel K. L. E. (Ed.) *Interval Mathematics*, Academic Press, New York, 1980.

[10] Older W. and Vellino A. "Extending Prolog with Constraint Arithmetic on Real Intervals" Proceedings of the Canadian Conference on Electrical and Computer Engineering, 1990.

[11] van Hentenryck, P. *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.