Extending Prolog with Constraint Arithmetic on Real Intervals *

William Older
André Vellino
Computing Research Laboratory
Bell-Northern Research
P.O. Box 3511, Station C
K1Y 4H7, Ottawa, Ontario

Abstract

Prolog can be extended by a system of constraints on closed intervals to perform declarative relational arithmetic. Imposing constraints on an interval can narrow its range and propagate the narrowing to other intervals related to it by constraint equations or inequalities. Relational interval arithmetic can be used to contain floating point errors and, when combined with Prolog backtracking, to obtain numeric solutions to linear and non-linear rational constraint satisfaction problems over the reals (e.g. n-degree polynomial equations). This technique differs from other constraint logic programming (CLP) systems like CLP($\Re$) or Prolog-III in that it does not do any symbolic processing.

1 Introduction

In most programming languages arithmetic operations are performed by a functional evaluation mechanism. This is also true for logic programming languages such as Prolog even though such mechanisms are ill-suited to its generally relational nature. Arithmetic in Prolog is essentially no different from arithmetic in Fortran.

The functional nature of Prolog arithmetic implies that arithmetic expressions will not, in general, commute with one another, e.g. `Y is 2 * Z, X is Y + B` is not, in general, equivalent to `X is Y + B, Y is 2 * Z`. Since "," (comma) is interpreted in Prolog as the logical "and" relation, this non-commutativity of arithmetic operations is a major weakness in the logical structure of the Prolog language.

The above problems apply even to integer arithmetic; when floating point arithmetic is considered there is an additional (and familiar) problem. Because of the representation limitations of floating point arithmetic, real quantities are not generally exactly representable in the internal binary formats, and rounding errors in the basic arithmetic operations cause the results to differ from those of rational arithmetic. As a result, floating point arithmetic no longer satisfies the axioms for real numbers, and one observes logical anomalies such as the failure of the equalities `1.21 == 1.1 * 1.1` and `(X + Y) + Z == X + (Y + Z)` for some X, Y and Z. (`==` is the Prolog arithmetic equality operator.)

However small these errors may be, numerically speaking, the result is that the symbolic manipulation of expressions according to the algebraic axioms of real arithmetic is not safe in the context of conventional floating point arithmetic.

These problems are, of course, not new ones, but have plagued numerical computation since the earliest days of digital computers. One mechanism for overcoming these problems by applying a Prolog-like mechanism to intervals was first suggested by Cleary [4]; these ideas were first fully implemented at Bell-Northern Research (BNR) in BNR Prolog in 1987 and have been successfully applied to problems far more complex than those described in [4]. Although similar in intent, Cleary's mechanism differs substantially from other constraint logic programming languages such as CLP($\Re$) [9], Prolog-III [5] and CHIP [7] in that it does not use term-rewriting or symbolic equation solving techniques. In many repects the interval constraint system of BNR Prolog most closely resembles the one described by Hyvönen in [8].

In the remainder of this paper we briefly describe the underlying mechanism behind constraint interval arithmetic and provide an overview of its salient features.

2   Relational Interval Arithmetic

One of BNR Prolog's experimental features is a system for treating arithmetical relations over the reals that is distinct from the usual evaluation mechanism for functional arithmetic. Since the system effectively has two inference engines for arithmetic, there needs to be a means for determining which executable statements are to be handled by which engine. The mechanism for doing this is a new primitive type called an *interval* and instances of intervals are introduced by range "declarations" which look like:

```
range(Quantity, [Lowerbound, Upperbound])
```

where `Quantity` is a logic variable which becomes instantiated to the newly created interval object, and `Lowerbound` and `Upperbound` are floating point numbers which define initial limitations on the range of `Quantity`. Subsequent actions may (and usually do) further *narrow* the range.

Whenever an arithmetic statement referring to interval objects is encountered, it is passed to the relational interval arithmetic subsystem for processing. Arithmetical statements are those stating equalities (`==`), inequalities (like `=<` and `>=`) and statements of the form `Variable is Expression` which define new interval quantities as arithmetic functions of existing interval quantities.

The relational interval arithmetic subsystem can be thought of as a specialized inference engine, for reasoning about the order properties of real numbers that is integrated with a constraint management system. The system retains a representation of all the arithmetic constraints so far imposed and the current best estimate of range restrictions on all intervals. The addition of an arithmetic statement augments the constraint set and may cause the range estimates for all intervals to narrow as the constraints are propagated through the network. If an inconsistency is discovered, failure is reported to the Prolog system, which backtracks to the last choice point. Careful control of rounding direction is required to ensure that an inconsistency is never falsely reported.

There are two interpretations of what an interval represents. Under the *numeric* interpretation, an interval represents an unknown real number that lies somewhere between the upper and lower bounds. Under the *regional* interpretation, an interval is the set of all points between its upper and lower bounds. This is analogous to the two interpretations of a logic variable: one where the variable refers to something specific (but unknown) and the other where the variable refers to the set of all its valid instantiations.

An interval whose upper and lower bounds are completely unknown, an indefinite interval, is represented as having the bounds given by the largest positive and largest negative interval numbers represented by the system, which we will denote by $-\infty$ and $+\infty$ respectively.

As forward computation proceeds, intervals can narrow by raising their lower bounds, lowering their upper bounds, or both. Narrowing occurs when additional constraints are applied to an interval, much as unifications increase the degree of instantiation of logic variables. Backtracking may undo the narrowing of an interval in much the same way that an ordinary variable is unbound.

For example, the simple query

```
?- range(X, [2.5, 6.3]), (X >= 3.9, X =< 4.7)
                       ; (X >= 2.8, X =< 3.5).
```

first narrows `X` to the solution [3.8999,4.7001] and on backtracking through the ";" (Prolog "or") narrows `X` to the alternative solution [2.7999, 3.5001].

2.1 Constraint Propagation

What happens when an arithmetic expression containing intervals is evaluated? First it is transformed into a sequence of primitive interval operations which is then used to construct a constraint network similar to a data flow network. Data may flow (as a wave of "narrowing") through the network in any direction, but always from the more constrained to the less constrained variables. The undirected nature of the graph corresponds directly to its ability to represent relations and not just functions. The flow of data may be in a cyclic pattern corresponding to a fixpoint iteration, just as may occur in a spreadsheet system. At any time during the constraint propagation an inconsistency may be detected. As described above, this terminates the iteration and reports failure to the Prolog engine.

To address the problems of standard Prolog arithmetic outlined in the introduction, the primitive elements of the constraint network must satisfy three formal properties. First, interval variables can only narrow the ranges of their input intervals, i.e. intervals can never grow. Secondly, all the primitive constraint equations or inequalities should be idempotent operations, i.e. applying an operation twice is the same as applying it once. Third, these operations on intervals should be isotone, i.e. preserve set inclusion order. More precisely, if an interval which has an initial range R_i narrows to a final range R_f, then, any other initial range $Q_i \subseteq R_i$ will narrow to a final range Q_f such that $Q_f \subseteq R_f$.

When these three properties are satisfied by the individual primitive operations, arbitrary sequences of such operations will converge to results that are independent of the actual order of operations. As a consequence, the detailed scheduling of the primitive operations for execution is a matter of efficiency only, and is therefore a natural candidate for parallel processing. By lattice-theoretic arguments, one can show that the total operation will also be narrowing, idempotent, and isotone and that successive macro operations commute and satisfy an associative law (i.e. "," is both commutative and associative).

A class of issues alluded to in the introduction, those due to the limited precision of floating point arithmetic, is handled by the combination of two techniques. One is the use of outward rounding in all computations—the fundamental principle of functional interval arithmetic introduced in [11]—which ensures that all computed intervals are large enough to contain all "true" answers. The other is the use of interval intersection to implement equality. The idempotent, commutative, and associative laws of intersection correspond to the reflexive, symmetric, and transitive laws of equality, respectively. This combination ensures that the truth of all the algebraic laws of the real numbers are formally preserved. As a consequence, it becomes possible to employ safely both symbolic and numeric techniques in the same application, and thereby benefit from their complementary strengths.

This approach also has the effect that all detected inconsistencies have the force of a constructive proof that no solution exists (over the mathematical reals) for the stated problem. However, if no inconsistency is detected, the computed narrowings

must be given a conditional interpretation: if solutions exist given the initial interval, they must be contained in the output intervals.

2.2   Equality and Inequality

One way to narrow an interval is by constraining it to be equal to another interval. If two intervals `range(X, [Xl, Xu])` and `range(Y, [Yl, Yu])` are constrained by equality, `X==Y`, then both `X` and `Y` are narrowed to the interval `[max(Xl,Yl), min(Xu,Yu)]`. In other words equating the intervals `X` and `Y` intersects them and imposes that constraint downstream. This notion of equality should be contrasted with that of equality in functional interval arithmetic where two intervals are equal if and only if their bounds are the same. Of course, if the ranges of `X` and `Y` do not intersect then equating them fails, thus forcing backtracking. The expression `not(X==Y)` succeeds if `X` and `Y` have no points in common and does no narrowing of either `X` or `Y`. In effect, `not(X==Y)` can be used to test that two intervals are disjoint.

Equating two intervals can be understood in either the numeric or the regional interpretation. The equality constraint between `X` and `Y` can be read as either "the real number denoted by the interval `X` is the same as the real number denoted by the interval `Y`" or as "the region spanned by `X` is the same as the region spanned by `Y`".

Another way to narrow intervals is by evaluating inequality relations. If $X = [X_l, X_u]$ and $Y = [Y_l, Y_u]$ the narrowing obtained by the constraint $X =< Y$ is computed by intersecting X with $[-\infty, Y_u]$ and Y with $[X_l, +\infty]$. For example, if X=[2,4] and Y=[1,6] and the expression $X =< Y$ is evaluated then Y is narrowed to [2,6] and X is not narrowed.

2.3   Arithmetic Operations

All the basic arithmetic operations can be used in expressing interval constraints. To do this, we first need to define the basic arithmetic functions on intervals. For any binary functional operation $\odot \in \{*, +, \div, -\}$ on intervals $[X_l, X_u]$ and $[Y_l, Y_u]$, we can define its functional evaluation to the interval $[A, B]$ by

$$[A, B] \leftarrow [X_l, X_u] \odot [Y_l, Y_u]$$

where

$$A \leftarrow glb\{x \odot y | x \in [X_l, X_u] \text{ and } y \in [Y_l, Y_u]\}$$
$$B \leftarrow lub\{x \odot y | x \in [X_l, X_u] \text{ and } y \in [Y_l, Y_u]\}$$

For the implementation of this idea, it is necessary to consider all the special cases (such as division for intervals that span zero). But in the simplest case, where $\odot$ is continuous and monotone in both variables,

$$[A, B] \leftarrow [X_l, X_u] \odot [Y_l, Y_u]$$

5

where
$$A \leftarrow min_{\downarrow}(X_l \odot Y_l, X_l \odot Y_u, X_u \odot Y_l, X_u \odot Y_u)$$
and
$$B \leftarrow max^{\uparrow}(X_l \odot Y_l, X_l \odot Y_u, X_u \odot Y_l, X_u \odot Y_u)$$

Since a computed operation $\odot$ causes rounding errors the min and max functions must round down ($\downarrow$) and up ($\uparrow$) respectively.

Now we can define all the interval relations in terms of the functional operations. For example the relational equation `X + Y == Z` is computed by

$$[Z_l', Z_u'] \leftarrow \cap\{[Z_l, Z_u], [X_l, X_u] + [Y_l, Y_u]\},$$

$$[X_l', X_u'] \leftarrow \cap\{[X_l, X_u], [Z_l, Z_u] - [Y_l, Y_u]\},$$

$$[Y_l', Y_u'] \leftarrow \cap\{[Y_l, Y_u], [Z_l, Z_u] - [X_l, X_u]\}.$$

where the primed bounds refer to the values for the intervals after the relational equation is evaluated and $\cap$ is the function that returns the intersection of two intervals. It follows from this definition of relational addition that the evaluation of a relational equation may well narrow all the intervals in it. For example evaluating the equation `X + Y == Z`, for initial values X=[3,7], Y=[2,8] and Z=[4,6] narrows all three intervals: X to [3,4], Y to [2,3] and Z to [5,6][1].

2.4   Critical Path Analysis

The following example solves a simple critical path analysis (PERT) problem. Let the time required to complete an activity, the *activity time*, be the difference between the start time and the finish time. Let the *slack time* be the difference between the latest and the earliest times an activity can be completed without disrupting the project. The *critical path* is then the list of activities between start and finish which have no slack time. Now we show how to formulate a critical path scheduling problem using equality and inequalities on intervals.

Figure 1 shows a network of activities (a-g) labeled by their activity times. Given that the Start time is 0 and that the Total time is unknown (but constrained to be less than the deadline), planning is done by constraining each start and finish time for each activity to be `Start + Duration == Finish`, and then constraining each start and finish time for each activity to obey the precedence orderings defined by the digraph. The resulting critical path is marked by the bold arrows.

```
?- create_intervals([AFinish, BFinish, CFinish,
              DFinish, EFinish, FFinish, GFinish,
```

___
[1]For this example the declarative reading of `X + Y == Z`, in the numeric interpretation is "the sum of some point in [3,4] and some point in [2,3] is equal to some point in [5,6]". Note, however, that if, downstream X narrows to the point [4,4] and Y to [3,3] then `X + Y == Z` is now false and such a narrowing forces failure.
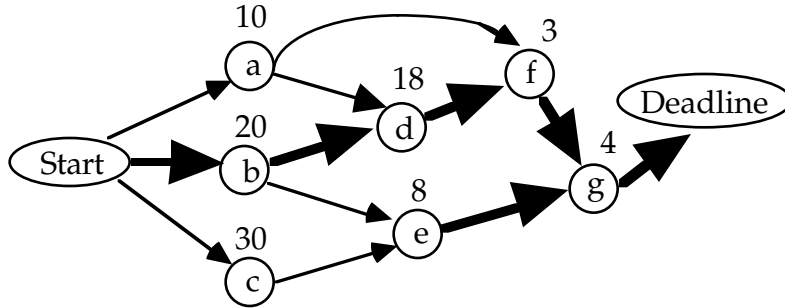
Figure 1: Critical Path Scheduling Problem

```
              Total, AStart, BStart, CStart,
              DStart, EStart, FStart, GStart]),
  Start = 0, Deadline = 50, Total =< Deadline,
  AFinish == AStart + 10, BFinish == BStart + 20,
  CFinish == CStart + 30, DFinish == DStart + 18,
  EFinish == EStart + 8, FFinish == FStart + 3,
  GFinish == GStart + 4,
  Start =< AStart, Start =< BStart, Start =< CStart,
  AFinish =< DStart, BFinish =< DStart,
  BFinish =< EStart, CFinish =< EStart,
  AFinish =< FStart, DFinish =< FStart,
  EFinish =< GStart, FFinish =< GStart,
  GFinish =< Total.
% create_intervals/1 recursively applies the
% range declaration to a list of variable.
create_intervals([]).
create_intervals([I|Is]) :-
              range(I,[_, _]),
              create_intervals(Is).
```

The narrowing of the activity times for each activity is performed simply by the inequality constraints on the start times of each activity and the equality constraints given by activities; no further algorithm is necessary.

2.5   Numerical Equation Solving

Some systems of simultaneous equations, linear and non-linear, can be solved using the narrowing induced by the constraint network alone. However, since the constraint satisfaction mechanism for narrowing intervals is based on local propagation (and not on term-rewriting), it is not sufficient on its own to solve even simple sets of linear equations unless they are in triangular form. This well known limitation of local propagation methods can be overcome with the use of additional techniques

described in [2] and [4].

In essence these techniques involve the formulation of explicit algorithms for exploring the validity of subdivisions for the intervals in the constraint network. Consequently, different algorithms may be better suited to different problems. In particular, it is possible to minimize the amount of work that the interval constraint management system does for specific problems. For instance, the problem of finding roots of a function $Y == f(X)$ can be made more efficient by a search procedure that implements a generalized Bernoulli method for subdividing intervals. The method consists in subdividing sub-ranges of X only if the interval Y contains 0 but without actually imposing the constraint that it does until the subdivisions have been done. The generic solver for BNR Prolog takes 2 seconds on a Mac-II to find all three roots for the equation $35x^{256} - 14x^{17} + x = 0$.

3   Comparisons with other Systems

Other constraint logic programming systems such as CLP($\Re$) [9], Prolog-III [5], Bertrand [10], CHIP [7] and Trilogy [1] all solve constraint satisfaction problems differently from BNR Prolog. Most of the systems that contain equation solvers usually operate only on systems of linear equations, or in the case of CLP($\Re$) delay the evaluation of non-linear equations until such time as they are sufficiently constrained that they become linear. Systems like CHIP and Trilogy, impose constraints either on finite sets or on integer- valued intervals. (For a more detailed comparison of some of these systems with BNR Prolog, see [3].)

BNR Prolog, on the other hand, has focused on the use of local propagation techniques on constraint networks of closed intervals on the real line (see [6]). This has the advantage over some other CLP systems in that the domain of application is extended to non-linear problems on the reals, not just to linear problems (see [12] for examples of optimization and equation solving problems using interval arithmetic). However, the problems for which local constraint propagation is not particularly well suited, such as linear programming, are not solved efficiently with divide and conquer algorithms. Moreover, there may be situations which require that a complex set of equations be reduced to a simpler set, for which a term rewriting system is preferable. Indeed, BNR Prolog could be augmented by a symbolic solver which could simplify the constraint network for the numerically based interval arithmetic subsystem.

4   Conclusions

Relational arithmetic on intervals makes it possible to express arithmetic logically in the framework of a logic programming language such as Prolog. It can also be used to contain floating point errors and to solve simultaneous non-linear equations

by systematically narrowing constrained intervals. In addition, the relational character of interval arithmetic encourages an economy of coding: inverse functions, for example don't need to be defined explicitly. But most importantly, the preservation of algebraic properties of expressions in interval arithmetic makes it safe to combine symbolic techniques with numerical techniques because the same expressions can be used either to evaluate numeric or symbolic values. Interval arithmetic operations are declarative, logical and correct.

References

[1] Andrews, J. *Trilogy User Manual*, Complete Logic Systems Inc. 1987.

[2] BNR Prolog *User Guide and Reference Manual*, BNR 1988.

[3] 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.

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

[5] Colmerauer, A. "Opening the Prolog-III Universe", Byte Magazine, August 1987.

[6] Davis, E. "Constraint propagation with interval labels", *Artificial Intelligence*, **32** pp. 281–331, 1987.

[7] Dincbas M., Simonis H. and van Hentenryck P. "Soving Large Combinatorial Problems in Logic Programming", *Journal of Logic Programming* **8**, 1&2, pp. 72-93, 1990.

[8] Hyvönen, E. (1989) "Constraint Reasoning Based on Interval Arithmetic", Procedings of *IJCAI 1989* pp. 193–199.

[9] 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.

[10] Leler, W. *Constraint Programming Languages their specification and generation*, Addison-Wesley pp. 202, 1988.

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

[12] Moore, R. E. (Ed.) *Reliability in Computing (The role of Interval Methods in Scientific Computing)*, Perspectives in Computing, **19**, Academic Press 1988.