Programming in CLP(BNR)

William J. Older
Bell Northern Research
Computing Research Laboratory
PO Box 3511, Station C
K1Y 4H7 Ottawa, Ontario, Canada

Frédéric Benhamou
Groupe Intelligence Artificielle
Faculté des Sciences de Luminy
case 901,163, avenue de Luminy
13288 Marseille Cedex 9 France
benham@gia.univ-mrs.fr

Abstract

CLP(BNR) is a constraint system based on relational interval arithmetic and forms part of BNR Prolog /v.4. This is a much more powerful system than previous versions and allows a much wider class of problems to be handled, including discrete domain problems and boolean equations. It is also integrated more closely into Prolog, thus providing for smoother and more flexible interaction between Prolog and the constraint system. This paper describes the programming interface and gives some simple programming examples.

1 Introduction

The problems of providing a logical form of arithmetic for use in Prolog are well-known. The difficulties of doing correct computations with floating point are also well-known. One mechanism for overcoming both of these problems — applying a Prolog-like narrowing mechanism to intervals — was first suggested by Cleary [Cleary 1987]. These ideas were first fully implemented at Bell-Northern Research (BNR) in BNR Prolog in 1987 [BNR Prolog 1988] and have been successfully applied to problems far more complex than those described in [Cleary 1987]. Although similar in intent, Cleary's mechanism differs substantially from other constraint logic programming languages such as CLP($\Re$) [Jaffar and Michaylov 1987] and Prolog-III [Colmerauer 1990] in that it is not based on term-rewriting or symbolic equation solving techniques. In many respects the interval constraint system of BNR Prolog most closely resembles CHIP [Dincbas, Simonis and Van Hentenryck 1990] in so far as it is based primarily on local propagation techniques, and the system described by Hyvonen in [Hyvonen 89] which also handles continuous quantities.

A general description of the technique of relational interval arithmetic and its abstract semantics is given in [Older and Vellino 1993]. Briefly, the description of the problem given by the user is compiled into a constraint network whose nodes are instances of the primitive relations supported by the system. The variables of the problem are associated with a set (regarded as a state) determined by their upper and lower bounds; the states are partially ordered by set inclusion. The constraint network then defines an operator on states which is monotone and contracting with respect to the partial order on states, and which is also idempotent and correct. Correctness here means that no valid solution (e.g., in the theoretical real numbers) is ever eliminated during contraction; as a consequence of correctness the separation of multiple solutions must involve a mechanism such as backtracking or or-parallelism. This technique, since it works by removing non-solutions, can be regarded as a model elimination proof procedure, and this gives it quite different charateristics from constructivist exact arithmetic systems.

As shown in [Benhamou and Older 1992], a major advantage of this approach is that it provides a uniform treatment of a wide range of problems usually treated by very different methods. CLP(BNR) currently deals with boolean variables, the natural numbers, and the reals, and supports a large number of primitive relations. Because they share the same fundamental framework, it is possible to mix all three types of variables in a single problem. In the reals, it can deal with general non-linear functions, transcendental functions, and non-continuous functions on the same footing.

Because the underlying representation is based on intervals, this technique can also deal directly with the low-precision data and tolerance limits characteristic of most real-world applications. It is also for this reason an ideal tool for doing sensitivity analysis on complex models.

This paper is concerned solely with describing the user's view (and programming model) of CLP(BNR) by means of examples.

2 CLP(BNR)

CLP(BNR) is a sublanguage of BNR-Prolog for dealing with relations over the booleans, naturals, and reals. Because it is a separate sub-language, with its own notion of equality (==), it does not alter the usual syntactic meaning of Prolog unification (=). The two systems are closely coupled: both are relational, they share the notion of failure and backtracking, and bindings made by either system are visible in both. Prolog serves as a meta-language for CLP(BNR); in particular, since programming can be done in Prolog there is no need for programming constructs in CLP(BNR) proper.

The variables which are shared between the two systems are called intervals. An interval is a logic variable which has been constrained to only take numeric values (either integer or real) and is possibly further constrained to lie in a (closed) interval of the real line with (floating-point expressible) bounds. An interval can be created using syntax of the form:

```
V:real(LB,UB)   V:integer(LB,UB)   V:boolean
```

where `LB` and `UB` are either expressions which evaluate to numeric values or they are unbound variables, representing $\pm\infty$. Arithmetic relations on intervals, i.e., constraints, will either fail (in the usual Prolog sense), or will succeed, in which case the bounds of the intervals may have been changed (narrowed).

The following simple queries should give some feel for the CLP(BNR) system. (CLP(BNR) generally employs familiar syntax for arithmetic relations and functions, but details can be found in the Appendix.)

Establishing a constraint propagates information from the known to the less known

```
?- X:real(1,3),   Y**2==X.
   ==> X : real(1.0, 3.0),
       Y : real(-1.73205080756888, 1.73205080756888)
```

Here is the same relation with both integer arguments; note that `X` becomes bound:

```
?- X:integer(1,3), Y:integer, Y**2==X.
   ==> X : 1
       Y : integer(-1, 1)
```

If `Y` is constrained to be positive, then `Y` also becomes bound to its unique answer:

```
?- X:integer(1,3), Y:integer, Y**2==X, Y>0.
   ==> X : 1
       Y : 1
```

Similar rules apply to general boolean relations:

```
?- B:boolean, 1 == B and (C or ~D) .
   ==> B : 1
       C : boolean
       D : boolean

?- B:boolean,   1 == B and (C or ~D),   0 == B and C .
   ==> B : 1
       C : 0
```

In some cases where an equation has a unique solution, equation solving is automatic:

```
?- [X,Y]:real, 1 == X + 2*Y,  Y - 3*X == 0.  % pair of linear eqns.
   ==> X : real(0.142857142857143, 0.142857142857143)
       Y : real(0.428571428571429, 0.428571428571429)
```

Here is a more interesting example, but with non-linear (including transcendental) equation solving:

```
?- [X,Y]:real,X>=0,Y>=0,  tan(X) == Y, X**2+Y**2 == 5.
   ==>  X : real(1.09666812870547, 1.09666812870547)
        Y : real(1.94867108960995, 1.94867108960995)
```

Note that although the upper and lower bounds in these answers print the same at this printing precision, the internal binary forms must differ by at least one bit in the last place, or else the variables would have been bound to the exact answer.

For more complex problems, which may have multiple solutions, there is a "solve" predicate which separates the solutions (by backtracking) and forces convergence. For example, for polynomial root finding :

```
?- X:real(0,1), 0== 35*X**256 - 14*X**17 + X, solve(X).
   ==>  X : 0.0                                          % 1st sol.

   ==>  X : real(0.847943660827315, 0.847943660827315) % 2nd sol.

   ==>  X : real(0.995842494200498, 0.995842494200498) % last sol.
```

Similarly, for a pair of simultaneous non-linear equations :

```
?- [X,Y]:real,  X**3 + Y**3 ==2*X*Y, X**2+Y**2==1, X>=0, solve(X).
   ==> X : real(0.391018886096038, 0.391085781049752)
   Y : real(-0.920382654506382, -0.92035423172858)

   ==> X : real(0.449060394395367, 0.450226789190836)
       Y : real(0.892914239048135, 0.893501405810577)

   ==> X : real(0.892906985142645, 0.893513338815017)
       Y : real(0.449036650353057, 0.450241175242194)
```

These examples have all been pure CLP(BNR) problems. The next few examples use a mixture of Prolog and CLP(BNR), although they can, if one chooses, equally well be regarded as programming in CLP(BNR).

3 Scheduling

The following example of a critical path algorithm computes a minimal completion time for a collection of activities with precedence relations[1]. First we create a "language" for describing precedence relations between activities:

```
op(700, xfx, before). % the precedes relation

activity(Name, Duration, task(Name,Start, Finish) ) :-
  [Start, Finish] : real,
  Finish == Start + Duration.

task(_,_,Finish) before task(_,Start,_):- Finish =< Start.
```

A particular scheduling problem can then be easily formulated, e.g.:

[1] Note this is scheduling over the Reals.

```
project( Start, Finish, [A,B,C,D,E,F,G]):-
  activity(a,10,A), Start before A,
  activity(b,20,B), Start before B,
  activity(c,30,C), Start before C,
  activity(d,18,D), A before D, B before D,
  activity(e, 8,E), B before E, C before E,
  activity(f,3,F), D before F,
  activity(g,4,G), F before G,
  G before Finish.
```

The actual computation of the shortest completion time and the start and finish windows for each task is then done by:

```
?- activity(start,0,S), activity(end,0,E), project(S,E,List),
   lower_bound(E).
```

3.1 Disjunctive Scheduling (Resource Limits)

Boolean constraints in BNR Prolog can be used not just for expressing pure boolean constraint problems (e.g. digital circuits), but are also useful in mixed computations, such as for disjunctive scheduling problems. For example, suppose that `S` is the start and `D` the duration of the task `T`. Now suppose that `T1` and `T2` (beginning at `S1, S2` and of duration `D1, D2` respectively), which are subject to precedence constraints, share the same resource and cannot execute concurrently (i.e. must be scheduled disjunctively). This constraint could be expressed by the formula:

```
(S1 + D1 =< B2) xor (B2 + D2 =< B1) == 1
```

where the `xor` constraint expresses the fact that the constraints are exclusive.

This way of expressing the problem provides an efficient alternative to the standard Prolog way of expressing disjunctive constraints by introducing a choicepoint. It can be applied effectively to solve scheduling problems with mutually exclusive resource allocation such as the bridge problem described in [Van Hentenryck 1989]. The CLP(BNR) program for this problem, which involves forty-six tasks and more than six hundred constraints, is described in detail in the appendix to [Benhamou and Older 1992].

4 Magic Series

Mixed boolean and integer constraints can be used to solve puzzles like the magic series [Colmerauer 1990, Van Hentenryck 1989]. The problem here is to find a sequence of non-negative integers $(x_0, ...x_{n-1})$ such that, for every i in $\{0, ..., n-1\}$, x_i is the number of occurences of the integer i in the sequence. In other words, for every $i \in \{0, ..., n-1\}$

$$x_i = \sum_{j=0}^{n-1} (x_j = i),$$

where the value of $(x = y)$ is 1 if $(x = y)$ is true and 0 if $(x \neq y)$ is true. Moreover, it can be shown that the two following properties are true.

$$\sum_{i=0}^{n-1} x_i = n, \sum_{i=0}^{n-1} i\, x_i = n$$

The BNR Prolog program that expresses these constraints is as follows:

```
magic(N,L) :-
  length(L,N),
  L : integral(0,_),
  constraints(L,L,0,N,N),
```

```
  firstfail(L).

constraints(L,[],N,0,0).
constraints(L,[X|Xs],I,S,S2) :-
  sum(L,I,X),
  J is I + 1,
  constraints(L,Xs,J,S1,S3),
  S==X + S1,
  S2 == X* I + S3.
  sum([],I,0).

sum([X|Xs],I,S) :-
  sum(Xs,I,S1),
  S== (X == I) + S1.
```

Thus the query that computes the magic series of length 4 produces the following results:

```
?- magic(4,L)
?- magic(4, [1, 2, 1, 0]). ;
?- magic(4, [2, 0, 2, 0]). ;
```

5 Non-Linear Continuous Constraints

This example is adapted from [Hong 1993] and serves to illustrate the application of interval arithmetic to continuous non-linear problems. (This case is equivalent to a pair of fourth degree polynomial inequalities.) This problem involves hitting a bird of known size and trajectory with a stone which has a fixed initial velocity and can only be launched at certain discrete instants. The problem is then to choose which instant and what elevation to use. The problem illustrates the logical nature of interval arithmetic, which makes it very natural in the context of Prolog.

The bird is modeled as a horizontal line segment of length `L`. The point `(X0,Y0)` is the bird initially and is located at `(0,H)`.

```
bird_init(X0,Y0,H,L):- X0>=0,X0=<L, Y==H.
```

The bird moves horizontally to the right with speed `U` and `(Dx,Dy)` is the displacement of the bird at time `T`.

```
bird_displacement(T,Dx,Dy,U):- Dx==U*T, Dy==0.
```

Then `(X,Y)`, the location of the bird at time `T`, is given by:

```
bird(T,X,Y,H,L,U) :-
  [Dx,Dy]:real,
   bird_displacement(T,Dx,Dy,U),
   bird_init(X-Dx, Y-Dy,H,L).
```

The projectile is modeled as a point; the (first) stone is initially located at `(0.0,0.0)`

```
stone_init(X,Y):- X==0.0,Y==0.0.
```

Note that the constants are not moved into the head of the clause; this will enable us to call `stone_init` with expressions for arguments later. (An expression, of course, will not generally unify with the constant `0.0`.)

Stones are shot with initial velocity `(Vx,Vy)`. The gravitational acceleration on earth is taken as `-9.8 m/sec*sec`, although an interval would be better. Let `(Dx,Dy)` be the displacement of the first stone at time `T`. Then the simplest formulation for the projectile motion is:

```
stone_displacement(T,Dx,Dy,Vx,Vy):-
  G is 9.8,
  Dx == Vx*T,
  Dy == Vy*T -  0.5*G*T**2 .
```

Then let `(X,Y)` be the position of a stone at time `T`:

```
stone(T,X,Y,Vx,Vy):-
  [Dx,Dy]:real,
  stone_displacement(T,Dx,Dy,Vx,Vy),
  stone_init(X-Dx,Y-Dy).
```

Stones are shot at time intervals of `Dt`; now let `(X,Y)` be the `N`-th stone at time `T`.

```
stone_N_th(T,X,Y,Vx,Vy,Dt,N):- stone(T-Dt*(N-1),X,Y,Vx,Vy).
```

The main procedure declares the basic variables and sets up the remaining constraints. The speed of shooting `V` is defined by `V**2==Vx**2 + Vy**2`; `S` is the slope of the initial stone trajectory and is restricted to be less than 1 (i.e. 45 degrees). Note that the last two calls equate the position of the bird and the position of the stone at the same time, i.e., the stone hits the bird.

```
hit(H,L,U,V,S,Dt, N, T,X,Y,Vx,Vy):-
  N:integer, % shot number
  Dt:real, % time between shots
  H:real, % height of bird
  L:real, % length of bird
  U:real, % x-velocity of bird
  V:real, % initial speed of stone
  S:real, % angle of stone trajectory when fired
  T:real, % time of hit
  [X,Y]:real, % position of stone
  [Vx,Vy]:real,% initial velocity of stone
  V**2 == Vx**2 + Vy**2, % magnitude of velocity is speed
  Vx >=0, Vy>=0,% only fire up and to right
  Vy== S*Vx, % slope restriction
  S >=0, S=<1.0,
  T>=0,  % only consider positive times
  X>=0, Y>=0, % and positions
  bird(T,X,Y,H,L,U),    % location of bird is same as
  stone_N_th(T,X,Y,Vx,Vy,Dt,N).   % location of stone
```

This initial formulation can now be run with specific data(bird 20 units high, second shot, etc.) :

```
?-hit(20.0,0.2,10.0,30.0,S,2.0,2,T,X,Y,Vx,Vy).
```

The answer (although a somewhat imprecise one) given by narrowing alone is then:

```
S : real(0.151327378188617, 1.0)
T : real(3.01569538493581, 7.4455056410137)
X : real(30.1518445255967, 75.9542677273962)
Y : real(20.0, 20.0)
Vx: real(5.53437068774039, 29.6622317367272)
Vy: real(4.59701024856432, 29.5097324297734)
```

The large size of these intervals can be reduced by several methods; the easiest method is to use solve on one of the variables:

```
?-hit(20.0,0.2,10.0,30.0,S,2.0,2,T,X,Y,Vx,Vy), solve(T).
```

This yields the much narrower answer:

```
S : real(0.878650132936296, 0.943989215521237)
T : real(3.63774689050188, 3.67373888991241)
X : real(36.3774689050188, 36.9373888991241)
Y : real(20.0, 20.0)
Vx: real(21.8153513340085, 22.5364861715658)
Vy: real(19.8016865705633, 20.5934563921109)
```

The size of the intervals is now determined mainly by the size of the bird. If the bird length is reduced to a point, these intervals also become near points, with the residual intervals reflecting round-off errors.

Another approach which is sometimes very useful is to add a redundant constraint

```
0 =< Vy - G*T
```

to the predicate `stone_displacement`. Although this merely expresses the fact that the stones's trajectory is everywhere lower than its highest point, even without solve it yields the answer:

```
S : real(0.878438127193421, 1.0)
T : real(3.59504189240291, 4.06175182228815)
X : real(35.9504189240291, 40.8175362967534)
Y : real(20.0, 20.0)
Vx: real(19.7989898732233, 22.5388553391693)
Vy: real(19.7989898732233, 22.5388553391693)
```

Finally, if we replace the formula

```
Dy== Vy*T - 0.5*G*T**2$$
```

in `stone_displacement` by the mathematically equivalent expression (formed by "completing the square")

```
Dy== (Vy*Vy)/(2*G) - (G/2)*(T - Vy/G)**2
```

we get — without solve — the answer:

```
S : real(0.895608876389984, 0.924134061383734)
T : real(3.60762768765977, 3.7015123584053)
X : real(36.0436531011464, 37.2740024982711)
Y : real(20.0, 20.0)
Vx: real(22.0296057795281, 22.352752887285)
Vy: real(20.0297702035754, 20.3540924671305)
```

Other questions, such as finding the solution(s) when the number of the stone is unknown, can also be easily formulated, e.g.:

```
?- N:integer(1,10), hit(20.0,0.2,10.0,30.0,S,2.0,N,T,X,Y,Vx,Vy),
   enumerate([N]).
```

Note that this problem involves both continuous and discrete variables.

6 Concluding Remarks

In problems involving pure integer and pure boolean constraints we have been able to compare the performance of the initial version of CLP(BNR) with other constraint systems specialized for these domains. The performance of CLP(BNR) appears to be roughly comparable to other systems (within a small linear factor) in most cases so far examined. The major exception is problems consisting mainly of not-equal relations, for which interval representations are not well suited.

One of the advantages of this approach is that the uniform semantics makes it easy to handle cases involving a mixture of integers, booleans, and reals in the same problem. We have found that in many cases, such as disjunctive scheduling as discussed above, the use of mixed type interval formulations leads to significant performance improvements over previously available techniques, yet also simplifies the problem formulation.

One area of our ongoing research is that of continuous, non-linear constrained optimization problems, where this technology looks very promising. In marked contrast to conventional approaches, a useable general algorithm for this class of problems in BNR Prolog is a succinct direct encoding of the classical mathematical theory of such problems, and its execution provides in principle a formal proof of optimality as well as a numerical solution.

The application of this technology to really complex problems is not always straightforward, because different formulations of the same problem can sometimes have quite different performance characteristics. Finding "good" formulations is therefore still a process of discovery, guided by experience, intuition, and a handful of basic principles, such as the deferring of choices, the first fail principle, and the controlled use of redundancy. Once found, however, such formulations are usually transparently clear (but possibly subtle) statements of the problem.

Acknowledgements

The authors wish to acknowledge the contributions of R. Workman, J. Rummell, and A. Vellino of Bell Northern Research for their myriad contributions to this project.

References

[Benhamou and Older 1992] Benhamou F. and Older W. "Applying Interval Arithmetic to Real, Integer and Boolean Constraints" BNR Research Report, 1992.

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

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

[Colmerauer 1990] Colmerauer, A. "An Introduction to Prolog-III", Communications of the ACM, July 1990 p.70-90.

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

[Hyvonen 89] Hyvonen, E. (1989) "Constraint Reasoning Based on Interval Arithmetic", Procedings of IJCAI 1989 pp. 193–199.

[Jaffar and Michaylov 1987] 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.

[Hong 1993] Hoon Hong, "RISC-CLP(Real):Logic programming with Non- linear constraints over the Reals" in Constraint Logic Programming: Selected Research. F. Benhamou and A. Colmerauer, eds. MIT Press, 1993, to appear.

[Older and Vellino 1993] Older, W., "Constraint Arithmetic on Real Intervals" in Constraint Logic Programming: Selected Research. F. Benhamou and A. Colmerauer, eds. MIT Press, 1993, to appear.

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

[Van Hentenryck 1989] Van Hentenryck P. Constraint Satisfaction in Logic Programming, MIT Press, Cambridge, 1989.

Appendix

Table 1: Declarations and Definitions

Syntax	Name	Description
`X: Type(LB,UB)`	declaration	constrains logic variable(s) `X` to be of specified type and range
`X is Y`	definition	variable `X` takes the type of expression `Y`
`X := Y`	definition	same as `is` but only does interval arithmetic

Table 2: First-Order Relations

Syntax	Name	Description
`X == Y`	arithmetic equality	`X` and `Y` constrained to be equal
`X =< Y`	less than or equal	`X` is constrained to be less than or equal to`Y`
`X >= Y`	greater than or equal	`X` is constrained to be greater than or equal to `Y`
`X < Y`	strict less than	(may be unsound on real intervals)
`X > Y`	strict greater than	(may be unsound on real intervals)
`X <> Y`	dif, inequality	`X` and `Y` (both integer) are constrained to be distinct

Table 3: Second-Order Relations

Syntax	Name	Description
`X <= Y`	inclusion	`X` is constrained to be a subinterval of `Y`
`X |= Y`	start together	`X` and `Y` are constrained to have the same lower bound
`X =| Y`	end together	`X` and `Y` are constrained to have the same upper bound

Table 4: Dyadic

Operation	Type Signature	Restrictions
`Z := X + Y`	`I := I + I` or `R :=R + R`	
`Z := X - Y`	`I := I - I` or `R :=R - R`	
`Z := X * Y`	`I := I * I` or `R :=R * R`	
`Z := X / Y`	`R:= I / I` or `R:=R / R`	`Y=0, X<>0` automatically excluded
`Z := min(X,Y)`	`I:= min(I , I)` or `R :=min(R , R)`	
`Z := max(X,Y)`	`I:= max(I, I)` or `R :=max(R , R)`	
`Z := (X ; Y)`	`I := (I ; I)` or `R :=(R ;R)`	(means `Z==X or Z==Y`)

Table 5: Monadic

Operation	Type Signature	Restrictions
`Z := X ** N`	`I := I ** I ; R :=R ** I,`	`N` must be an integer constant
`Z := -X`	`I := -I ; R := -R`	
`Z := abs(X)`	`I := abs(I) ; R := abs(R)`	`Z >= 0`
`Z := sqrt(X)`	`R := sqrt(R)`	`Z>=0, X>=0`
`Z := exp(X)`	`R := exp(R)`	`Z>0`
`Z := ln(X)`	`R := ln(R)`	`X>0`
`Z := sin(X)`	`R := sin(R)`	`-1=<Z=<1`
`Z := asin(X)`	`R := asin(R)`	`-1=<X=<1,-pi/2=<Z=<pi/2`
`Z := cos(X)`	`R := cos(R)`	`-1=<Z=<1`
`Z := acos(X)`	`R := acos(R)`	`-1=<X=<1, -pi/2=<Z=<pi/2`
`Z:= tan(X)`	`R := tan(R)`	`-1=<Z=<1`
`Z := atan(X)`	`R := atan(R)`	`-1=<X=<1, -pi/2=<Z=<pi/2`

Table 6: Boolean

Operation	Type Signature
`Z := X and Y`	`B:= B and B` (and)
`Z := X nand Y`	`B:= B nand B`
`Z := X or Y`	`B:= B or B` (inclusive or)
`Z := X nor Y`	`B:= B nor B`
`Z := X xor Y`	`B:= B xor B` (exclusive or)
`Z :=  X`	`B:=  B` (boolean negation)

Table 7: Mixed

Operation	Type Signature	Restrictions
`Z := (X==Y)`	`B:=(R== R)`	test/impose equality
`Z := (X<>Y)`	`B:=(R<> R)`	equiv. `Z :=  (X==Y)`
`Z := (X=<Y)`	`B:= (R=<R)`	test/impose inequality
`Z := (X>Y)`	`B := (R>R)`	equiv. `Z :=  (X=<Y)`
`Z := (X>=Y)`	`B := (R>=R)`	equiv. `Z := (Y=<X)`
`Z := (X<Y)`	`B:=(R< R)`	equiv. `Z := (X>Y)`

Table 8: Miscellaneous

Syntax	Semantics
`lower_bound(X)`	interval `X` is narrowed to it lower bound
`upper_bound(X)`	interval `X` is narrowed to it upper bound

Table 9: Pseudo-functions on intervals

Syntax	Result
`midpoint(X)`	returns midpoint of an interval, i.e. `(UB-LB)/2`
`median(X)`	returns a point in an interval (suitable for splitting)
`delta(X)`	returns the width of an interval (i.e. `UB - LB`)

Table 10: Control and Enumeration Predicates

Predicate	Description
`interval(X)`	true if variable `X` is an interval, fails otherwise.
`domain(X, Type(L,U))`	get the type and bounds of interval `X`, fails otherwise.
`range( X, [L,U])`	queries the lower and upper bounds of an interval or numeric
`enumerate(IntervalList)`	systematic enumeration of a list of intervals (usually used on integer and boolean intervals).
`firstfail(IntegerIntervalList)`	systematic enumeration of a list of integer intervals in dynamic order of size.
`solve(X, N, Eps)`	a nondeterministic subdivision algorithm for real intervals which stops after `N` levels or when `delta(X)` becomes less than `Eps*(initial delta(X))`. `X` may also be a list of intervals.
`solve(X)`	the same as `solve(X,6,0.0001)`. Produces a maximum of 2**6 answers) (Note 3)
`presolve(X)`	an intelligent solve for use with complex problems with many variables
`absolve(X, N)`	deterministic predicate to trim the ends of interval `X` to make it as small as possible. `N` is a small integer (¡ 32) which controls the relative precision. `X` may be a list of intervals.
`absolve(X)`	same as `absolve(X,14)`
`degrees_of_freedom(X,E,R,I,D)`	returns the number of equations `E`, continuous variables `R`, inequalities `I`, and discrete (integer or boolean) variables `D` in the constraint network attached to interval `X`.
`constraint_network(X, N where S)`	returns a description of the network attached to interval `X` and its current state.