To appear in *Computational Intelligence* **8**(4), Blackwell Publishers., 1992

Hierarchical Arc Consistency for Disjoint Real Intervals in Constraint Logic Programming

Greg Sidebottom and William S. Havens

Expert Systems Lab
Centre for Systems Science and
School of Computing Science
Simon Fraser University
Burnaby, British Columbia, V5A 1S6, Canada
Phone: (604) 291-4623
Email: {gregory,havens}@cs.sfu.ca

Abstract: There have been many proposals for adding sound implementations of numeric processing to Prolog. This paper describes an approach to numeric constraint processing which has been implemented in Echidna, a new constraint logic programming (CLP) language. Echidna uses consistency algorithms which can actively process a wider variety of numeric constraints than most other CLP systems, including constraints containing some common non-linear functions. A unique feature of Echidna is that it implements domains for real-valued variables with hierarchical data structures and exploits this structure using a hierarchical arc consistency algorithm specialized for numeric constraints. This gives Echidna two advantages over other systems. First, the union of disjoint intervals can be represented directly. Other approaches require trying each disjoint interval in turn during backtrack search. Second, the hierarchical structure facilitates varying the precision of constraint processing. Consequently, it is possible to implement more effective constraint processing control algorithms which avoid unnecessary detailed domain analysis. These advantages distinguish Echidna from other CLP systems for numeric constraint processing.

Key words: Constraint logic programming, numeric constraint processing, constraint-based reasoning, hierarchical arc consistency, partial arc consistency.

Subject Categories:
Major: *Logic Programming, Languages/tools*
Minor: *Theorem Proving*

1. Introduction

There have been many proposals for adding sound implementations of numeric processing to Prolog. Constraint logic programming (CLP) languages like CLP(ℝ) (Jaffar and Michaylov, 1987), Prolog III (Colmerauer, 1990), and CAL (Aiba et al., 1988) use symbolic constraint solving techniques. However, CLP(ℝ) and Prolog III can only actively process[1] linear constraints using linear programming algorithms. CAL actively processes polynomial constraints using algorithms from polynomial ideal theory which have doubly exponential time complexity in the worst case (Buchberger, 1985).

The CHIP CLP language (Van Hentenryck, 1989) and BNR Prolog (Older and Vellino, 1990) use consistency and case analysis algorithms (Mackworth, 1977) for solving constraints. Consistency algorithms require that variables be associated with domains which are sets of candidate values for a variable. For consistency algorithms, a domain must be represented by some finite manipulatable structure. CHIP's numeric domains are always finite integer sets[2]. BNR Prolog's domains are real intervals and it efficiently implements many constraints on real numbers by using consistency algorithms to tighten those intervals closer to actual solutions to the constraints (Cleary, 1987).

To compare numeric CLP systems which use consistency/case analysis algorithms, it is useful to distinguish between solutions to a query and answers to that query given by a CLP language. A solution for a query Q with respect to a program P is a substitution s of terms for Q's variables such that sQ is a logical consequence of P with each constraint given its usual interpretation in the real number system. Many practical consistency algorithms only partially solve CSPs, restricting domains to sets of values which are only locally consistent. When domains are not finite, case analysis algorithms may be insufficient to completely solve the CSP under consideration. Thus, in some cases CLP systems may give answers which are not solutions to queries. It is reasonable to expect, however, that the set of answers given to a query is a super set of the solutions to that query. That is, no solutions are missed.

Both CHIP and BNR Prolog provide case analysis primitives which can be used to augment consistency algorithms within the logic programming (LP) language. CHIP can answer a query with exactly its set of solutions since its consistency/case analysis algorithms only deal with finite discrete sets. Case analysis methods in BNR Prolog can tighten intervals as close to solutions as possible using the underlying finite precision floating point computer arithmetic. This ensures that no solution for a given set of constraints is missed although sometimes answers contain no solutions.

This paper describes an approach to real number constraint processing which has been implemented in a new CLP language called Echidna (Havens, et al., 1990; Havens, 1991). Like both CHIP and BNR Prolog, Echidna uses consistency algorithms which can actively

---

[1]By active constraint processing, we mean using constraints to at least partially test their satisfiability when they are encountered in the execution of a logic program. This is contrasted with the passive use of constraints which consists of testing them later once values for their arguments are generated.

[2]It should be noted that CHIP also processes linear constraints on rational variables with symbolic constraint processing algorithms. In this paper, we compare Echidna's algorithms only with CHIP's consistency/case analysis algorithms for finite discrete domain variables.

process a wide variety of real number constraints. The key difference is that real domains consisting of disjoint sets of intervals are implemented in Echidna with a hierarchical data structure. The language exploits this structure using a version of hierarchical arc consistency (Mackworth, Mulder and Havens, 1985) specialized for real number constraints.

Davis (1987) classifies constraint systems according to the richness of the language used to represent both variable domains (the "label language") and constraints (the "constraint language"). Echidna implements label and constraint languages which are more general than the real number systems surveyed in (Davis, 1987) and Echidna's label language is more general than BNR Prolog's. Our label language is a collection of disjoint real interval sets and it contains real intervals as a subset. This makes it possible for Echidna to efficiently deal with various constraints which lead to exponential search in BNR Prolog. Like BNR Prolog, our constraint language contains equalities, and inequalities on arbitrary expressions involving the arithmetic functions and trigonometric functions, and some expressions involving the exponential, root and logarithmic functions. Unlike BNR Prolog, our algorithms also support disjunctions of inequalities as atomic constraints.

Echidna handles such general systems of constraints by using *partial consistency* algorithms (Nadel, 1989). Partial consistency algorithms approximate the set of solutions to a constraint satisfaction problem (CSP) by computing a superset of the solutions. Good partial consistency algorithms compute a superset which is only slightly larger than the actual set but at a substantially reduced cost. Our method is easily parameterized to compute arbitrarily close to the actual set of solutions.

The remainder of the paper is organized as follows: Section 2 describes how Echidna augments a logic programming language with real number constraints. Section 3 briefly describes the SLD-resolution theorem prover (Lloyd, 1984) and arc consistency algorithms (Mackworth, 1977) used in Echidna. Section 4 specifies Echidna's adaptation of hierarchical arc consistency (Mackworth, et al., 1985) for real number constraints. Section 5 gives some sample runs using Echidna (version 1.0) and compares it with other CLP languages. Finally, Section 6 draws some conclusions about this research.

2. The Echidna Language

This section focuses on those aspects of Echidna concerned with real number processing. For a description of other aspects of the language, see (Havens, et al., 1990) and (Havens, 1991). In this presentation, we augment the syntax of Edinburgh Prolog (Sterling and Shapiro, 1986) as necessary for exposition.

Echidna provides domain constraints, equalities, inequalities, and disjunctions of inequalities on real numeric expressions. A domain constraint is a unary constraint of the form:

```
X ∈ Set
```

where `X` is a real valued variable and `Set` denotes the domain of `X`. The domain is specified as a finite union of open, closed, or half-open real intervals. An interval is specified by a lower and an upper bound. A bound consists of a real numeral and a bracket symbol. A square bracket indicates that the bound is closed and a round bracket indicates the bound is open, according to normal mathematical usage. For instance, `[0, 1]` denotes the set $\{x \mid 0 \leq x \leq 1\}$ and `[0, 1)` denotes the set $\{x \mid 0 \leq x < 1\}$. Intervals

which are not bounded above or below can be specified using the symbols `-∞` and `+∞`. For instance, `(0, +∞)` specifies the set of all positive real numbers.

The domain constraint:

```
X ∈ [0, 1] ∪ (3, 4] ∪ (7, 10)
```

declares `X` to be in the domain $\{x \mid 0 \le x \le 1 \vee 3 < x \le 4 \vee 7 < x < 10\}$.

Equalities and inequalities are constraints on real numeric expressions, henceforth referred to simply as *expressions*. Expressions are built up from variables and real constants using numeric function symbols[3]. The following is an example of an Echidna program using equalities and inequalities:

```
[1]  onCircle(p(X,Y), c(p(A,B), R)) :-
        R > 0,
        (X - A)^2 + (Y - B)^2 = R^2.
```

It specifies the relationship between a circle centered at point `p(A,B)` with radius `R` and a point `p(X,Y)` on its circumference, as shown in figure 1.

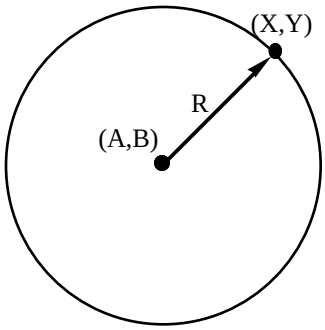

Figure 1. A circle described by [1]

The query:

```
[2]  ?- A ∈ [-100,100],
        B ∈ [-100,100],
        R ∈ [-100,100],
        C = c(p(A,B),R),
        onCircle(p(0,1), C),
        onCircle(p(1,0), C),
        onCircle(p(-1,0), C).
```

has a single solution:

---

[3]Echidna currently supports the arithmetic functions, some trigonometric functions, exponentiation, logarithm, and root extraction.

[3] `A = 0, B = 0, R = 1`

since there is a theorem from geometry which states that three points uniquely determine a circle. Notice that this query results in many constraints involving non-linear expressions. Echidna can restrict the domains of `A`, `B` and `R` to intervals which tightly bound this solution.

Echidna supports constraints of the form:

$E_1 \neq E_2$

where E_1 and E_2 are expressions, but these are a special form of the disjunctive inequality constraint which is written:

$C_1 \vee C_2$

where C_1 and C_2 are both inequalities. The disjunctive inequality is useful in temporal and spatial reasoning problems. For instance, Figure 2 gives an Echidna program for scheduling tasks using some of the relations on temporal intervals described in (Allen, 1983). A task is represented by a term `task(S, D)` where `S` is the start time of the task and `D` is the duration of the task. The predicate, `in(Task, SuperTask)`, is true if the interval for `SuperTask` contains the interval for `Task`. `NoOverlap(Task, Tasks)` is true if `Task` overlaps with none of the tasks in the list `Tasks`. It uses a disjunctive inequality constraint (shown in bold typeface in Figure 2) to make sure `Task` is either before or after all the tasks in `Tasks`. `Schedule(Tasks, SuperTask)` is true if all the tasks in the list `Tasks` are in `SuperTask` but no pair in `Tasks` overlap.

```
in(task(S1, D1), task(S2, D2)) :-
   S1 ≥ S2,
   S1+D1 ≤ S2+D2.

noOverlap(_, []).
noOverlap(task(S1, D1), [task(S2, D2) | Tasks]) :-
   S1+D1 ≤ S2 ∨ S1 ≥ S2 + D2,
   noOverlap(task(S1, D1), Tasks).

schedule([], _).
schedule([Task | Tasks], SuperTask) :-
   in(Task, SuperTask),
   noOverlap(Task, Tasks),
   schedule(Tasks, SuperTask).
```

Figure 2. An Echidna Program for scheduling tasks

Given the program of Figure 2, Echidna can deduce from the goal:

[4] `?- schedule([task(S1, 2), task(S2, 1.5)], task(0, 4)).`

that `S1` is in the set [0, 0.5] ∪ [1.5, 2] and `S2` is in the set [0, 0.5] ∪ [2, 2.5][4].

3. Overview of the Echidna Reasoning Engine

Echidna programs are executed by an SLD-resolution theorem prover (Lloyd, 1984) which incrementally constructs and maintains a CSP. A CSP is defined by a set of variables, each associated with a domain of candidate values and a set of constraints on subsets of the variables. A constraint specifies which values from the domains of its variables are compatible. The notation D_X is used to denote the domain of the variable `X`. For all variables `X` participating in real number constraints, D_X is a subset of the set $\mathbb{R}$ of real numbers. A *solution* to the CSP is an assignment of values to all its variables which satisfies all the constraints. For a CSP containing a variable `X`, a value $a \in D_X$ is *inconsistent* if it is not assigned to `X` in any solution to the CSP. When a constraint is selected by the theorem prover, it is added to the CSP. Echidna manipulates the CSP using two methods (Mackworth, 1977):

1. arc consistency is used to remove inconsistent values from the domains of variables under numeric constraints, and

2. heuristic case analysis is used to consider alternatively different halves of real variable domains[5].

If the arc consistency algorithm ever removes all values from a variable domain, then the constructed CSP has no solutions. The theorem prover then backtracks. Backtracking through a constraint consists of removing it from the CSP.

Case analysis provides a divide-and-conquer method for finding solutions to the CSP. Arc consistency is interleaved with case analysis algorithms to further reduce the search space. Case analysis is implemented by the built-in predicate, `split(Vars)`, which is similar to predicates described elsewhere (Cleary, 1987; Older and Vellino, 1990; Van Hentenryck, 1989). `Split(Vars)` repeatedly cycles through the list `Vars` of variables in a round-robin fashion removing approximately half the values in each variable's domain. Upon backtracking, `split` restores half of a domain and removes the other half.

Echidna's real constraint processing techniques are partial consistency and case analysis algorithms (Nadel, 1989) because they are not capable of completely solving the CSP. Arc consistency is not sufficient to solve CSPs because it considers only single constraints in isolation. When domains are finite, case analysis combined with arc consistency can completely solve the CSP. If Echidna's real domains were finite, then after some finite number of iterations, `split` would have reduced all variable domains to singleton sets. However, real domains are not finite. Currently, the number of times a variable domain is

---

[4]Actually, Echidna deduces sets slightly larger than these sets. See section 4 for details.

[5]Variables with finite domains, such as finite sets of integers, may also consider each value in the domain in turn. This is known as backtrack tree searching.

split is determined by a built-in predicate, `precision(Vars, Prec)`. Sections 4.2 and 5.1 describe its operation in more detail. We are investigating how to provide more flexible control of the case analysis methods.

4. Hierarchical Arc Consistency on Reals (HACR)

This section is organized as follows. Section 4.1 gives some notation for describing projections of constraints and describes the basic HACR algorithm in terms of this notation. Section 4.2 analyzes how domains are represented. Section 4.3 describes how Echidna's arc revision algorithm is implemented given the ability to compute the projections of constraints and section 4.4 describes how to compute projections for many constraints.

4.1 Constraint Notation and the Basic Algorithm

We use the notation $\nu(C)$ to denote the set of variables in an atomic constraint C. The arity of C is $|\nu(C)|$. If $\nu(C) = \{X_1,\ldots,X_k\}$, we sometimes write $C(X_1,\ldots,X_k)$ to make explicit the variables of C. We assume the CSP is formulated as a directed hypergraph[6] where variables are associated with nodes and each constraint C is associated with a set of hyperarcs of the form (T, C) for each $T \in \nu(C)$. T is called the *target* and the rest of the variables in $\nu(C)$ are called *sources*. Given a CSP of this form, an arc consistency algorithm deletes inconsistent values from target variable domains. These inconsistent values are such that there are no corresponding values for the source variables which satisfy the constraint. Such deleted values cannot be part of any global solution to the CSP. The notation Δ_X is used to denote the changing domain of the variable X which decreases monotonically from its full declared domain D_X towards smaller and smaller subsets. When values are deleted from Δ_X, it is said to be *refined*.

To simplify discussion, constraints are viewed as dynamically changing sets of mappings. Associated with each constraint $C(X_1,\ldots,X_k)$ is a relation $C^* \subseteq D_{X_1}\times\ldots\times D_{X_k}$ and $C = \{\{X_1\rightarrow a_1,\ldots,X_k\rightarrow a_k\} \mid (a_1,\ldots,a_k) \in C^* \cap \Delta_{X_1}\times\ldots\times\Delta_{X_k}\}$[7]. For example, if $\Delta_X = \Delta_Y = \{1, 2, 3\}$ and $C(X, Y)$ is the constraint that X is less than Y, then $C = \{\{X\rightarrow 1,Y\rightarrow 2\}, \{X\rightarrow 1,Y\rightarrow 3\}, \{X\rightarrow 2,Y\rightarrow 3\}\}$. What we mean by 'relations are viewed as sets of mappings' is that each element $\mu \in C$ can be considered a mapping $\mu:\nu(C) \rightarrow \bigcup_{x\in \nu(C)}\Delta_X$. For example, for $\mu = \{X\rightarrow 1,Y\rightarrow 2\} \in C$ above, $\mu(X) = 1$ and $\mu(Y) = 2$.

A value $a_i \in D_{X_i}$ can be used to *satisfy* a constraint $C(X_1,\ldots,X_i,\ldots,X_k)$ if and only if there exists $\mu \in C$ such that $\mu(X_i) = a_i$. C is *satisfiable* if and only if $C \neq \emptyset$.

---

[6] A directed hypergraph is a generalization of a directed graph where hyperarcs may 'connect' any number of nodes.

[7] One of the referees pointed out that C^* is actually the constraint on $\{X_1,\ldots,X_k\}$ and C is the structure manipulated by our algorithms. However, we believe that no confusion results if we continue to refer to C as a constraint.

A useful function for describing consistency algorithms is projection, denoted π, which maps a constraint C and a variable $X \in \nu(C)$ to a subset of Δ_X. It is defined by:

[5] $$\pi_X(C) = \{a \mid (\exists \mu \in C)\ \mu(X) = a\}.$$

For instance, given C = 'X < Y' as above, $\pi_X(C) = \{1, 2\}$ and $\pi_Y(C) = \{2, 3\}$. For any constraint C and any variable $X \in \nu(C)$, all values $a \in \Delta_X \setminus \pi_X(C)$[8] cannot be used to satisfy C since there are no corresponding values for $\nu(C) \setminus X$. Such values are *inconsistent with the constraint* C and thus cannot be part of any solution to the CSP. A hyperarc (T, C) is *arc consistent* if $\Delta_T = \pi_T(C)$. *Full* arc consistency algorithms delete all inconsistent values from every domain in the CSP, making all constraints arc consistent. *Partial* arc consistency algorithms (Nadel, 1989) delete only some inconsistent values. A well-designed partial arc consistency algorithm deletes most inconsistent values at less cost than any full consistency algorithm.

The fundamental operation of most arc consistency algorithms is arc *revision* (Mackworth, 1977), which is implemented by a procedure Revise(T, C) where (T, C) is a hyperarc. Revise refines Δ_T by deleting values which are inconsistent with C. *Full* arc revision is implemented by having Revise(T, C) perform the assignment $\Delta_T \leftarrow \pi_T(C)$, making the hyperarc (T, C) arc consistent. *Partial* arc revision sets Δ_T to some superset of $\pi_T(C)$.

Full arc consistency algorithms, such as AC-3 (Mackworth, 1977), call Revise repeatedly with various hyperarcs. These arc consistency algorithms terminate when there is no hyperarc (T, C) such that Revise(T, C) can refine Δ_T further. Echidna employs a similar but partial arc consistency algorithm, called HACR, for real number constraints. HACR repeatedly applies a partial arc revision algorithm, called ReviseHACR(T, C), to hyperarcs (T, C) thereby reducing Δ_T to some near superset of $\pi_T(C)$ which can be computed efficiently. HACR terminates when there is no hyperarc (T, C) such that ReviseHACR(T, C) can refine Δ_T further.

Figure 3 presents the HACR algorithm with an abstract specification of ReviseHACR. It is essentially the same as the AC-3 algorithm (Mackworth, 1977)[9], but it is generalized for *k*-ary constraints[10]. The input to HACR is a set A of hyperarcs which formulate the CSP. The CSP contains the constraints Echidna has selected during an SLD-derivation.

---

[8]The symbol '\' is used to denote set difference.

[9]It should be noted that there are newer asymptotically more efficient algorithms in the AC family: AC-4 (Mohr and Henderson, 1986) and AC-5 (Deville and Van Hentenryck, 1991).

[10]The initial step of achieving node consistency using the unary constraints has been removed, since the remainder of the algorithm handles unary constraints. However, it is usually most efficient to handle unary constraints first.

```
1   procedure HACR(A):

2       procedure ReviseHACRAbstract(T, C):
3       begin
4           let Δ be a set such that πT(C) ⊆ Δ ⊆ ΔT is true;
5           DELETE ← (Δ ⊂ ΔT);
6           if DELETE then ΔT ← Δ;
7           return DELETE
8       end;

9   begin
10      Q ← A;
11      while Q ≠ Ø do begin
12          select and delete any hyperarc (T, C) from Q;
13          if ReviseHACRAbstract(T, C) then
14              Q ← Q ∪ {(T', C') ∈ A | T ∈ ν(C')\{T'} ∧ C ≠ C'}
15      end
16  end;
```

Figure 3. HACR: an arc consistency algorithm for real constraints

The subprocedure, ReviseHACRAbstract, is an abstract specification of our partial arc revision algorithm, ReviseHACR. It specifies a partial arc revision algorithm because Δ, the new domain for the target variable T, is somewhere between Δ_T and $\pi_T(C)$, as specified on line 4. A good implementation of this specification makes Δ as close to $\pi_T(C)$ as efficiently possible. Lines 5 and 6 specify that Δ_T is updated only if ReviseHACRAbstract succeeds in refining it. ReviseHACRAbstract returns true if and only if Δ_T is refined. ReviseHACRAbstract's implementation depends on how domains are implemented and the class of constraints being processed.

Line 10 of HACR initializes Q to the set A of input hyperarcs. The loop from line 11 to line 15 removes and revises one hyperarc from Q in each iteration, so each hyperarc is revised at least once. If ReviseHACRAbstract(T, C) refines Δ_T in line 13, then Q is updated in line 14 to add just the set of hyperarcs which could be further revised. These are of the form (T', C') with $T \in \nu(C')\backslash\{T'\}$ and $C \neq C'$. This is because T is a source variable of C' so the partial arc consistency of some values in $\Delta_{T'}$ may have depended on values deleted from Δ_T. That is, $\pi_{T'}(C')$ may have changed since it depends on T. Hyperarcs involving the same constraint (C = C') are not added because (T', C) is such that T' is a source variable of the hyperarc (T, C) which was just refined. (T', C) cannot have become partially inconsistent because Δ_T was refined. Values were deleted from Δ_T precisely because there was no corresponding values for the source variables of (T, C).

Unlike arc consistency algorithms like AC-3 and HAC, which are for finite domains, there is no guarantee that full arc consistency algorithms for numeric domains terminate. This is because real domains can be refined indefinitely. Hence ReviseHACRAbstract must be a partial arc revision algorithm. Section 4.3 describes the built-in predicate, `precision`, which is used to limit domain refinement. HACR terminates when Q = Ø, the exit condition on line 11. Otherwise, the loop of lines 11-15 is executed. Line 12 deletes one hyperarc from Q. New hyperarcs are added to Q in line 14 after a domain is refined in line 13. At any point in an SLD-derivation, the number of variables and constraints in the CSP is finite. Thus, the number of domains is also finite. Since each of the domains will be refined only a finite number of times by ReviseHACRAbstract, at some point no hyperarcs will be added to Q. Thus, Q eventually becomes empty and HACR terminates.

4.2 Domains

Arc consistency algorithms usually operate on finite domains. Domains are represented extensionally as enumerated sets of candidate values. These algorithms can be very expensive when domain sizes are large. For instance, the running time of AC-3 is proportional to the square of the domain size in the best case and the cube in the worst case (Mackworth & Freuder, 1985). An extensional representation for real domains is impossible. Instead, we introduce a hierarchical and intensional domain representation.

HACR is based on the hierarchical arc consistency algorithm, HAC (Mackworth et al., 1985). HAC facilitates manipulating potentially very large discrete domains which can be organized as taxonomies. A taxonomy structures a domain into a hierarchy of subsets which have common properties and stand in common relations. HAC assumes that the taxonomies are relatively balanced and structured in a way appropriate for the constraints under consideration, and that all constraints are unary or binary. Under these assumptions, the running time of HAC is independent of domain size in the best case and is proportional to the logarithm of domain size in the worst case (Mackworth et al., 1985). Although HAC presumes that domains are finite, it actually manipulates domain subsets intensionally as symbols by precompiling predicates which test properties of these symbols. We describe this essential capability further in section 4.3.

Consider the example of Figure 4 taken from Mackworth et al. (1985). It shows taxonomies for the variables G and S where D_G is the set {*island*, *mainland*, *lake*, *ocean*} of geographic systems and D_S is the set {*lakeshore*, *coastline*} of shorelines. Each taxonomy is a directed tree with all arcs directed towards the leaves[11]. Each node is associated with a domain symbol, denoting a subset of the domain, and a mark. To be precise, we should distinguish between the symbol associated with each node and the domain subset which it denotes. However, ignoring the distinction is more convenient and does not result in confusion. Thus, we will refer to a node domain symbol simply as a node domain and manipulate it as if it were a set.

The arcs of the tree represent proper subset relations between node domains. The root domain is the full domain for the variable. The leaves are singleton subsets. Each child domain is a proper subset of its parent domain. The union of the children domains are

[11]It should be noted that hierachical arc consistency algorithms can generally operate on taxonomies which are rooted directed acyclic graphs. We only need to consider trees here.

assumed be equal to the parent domain and, for simplicity, the children domains are assumed to be disjoint.

Each node is associated with a mark indicating the relationship between its domain and the dynamic domain Δ_X for the variable X. Each sub-tree rooted at a particular node represents a particular subset of Δ_X. The mark for the root node of a sub-tree indicates whether its domain is completely contained in Δ_X (marked '√'), completely excluded from Δ_X (marked '×'), or partially contained in Δ_X (marked '?'). In the last case, the part of Δ_X represented by the sub-tree rooted at the node is union of the parts represented by the sub-trees rooted at its children. HAC maintains the dynamic domain Δ_X by manipulating these marks[12].

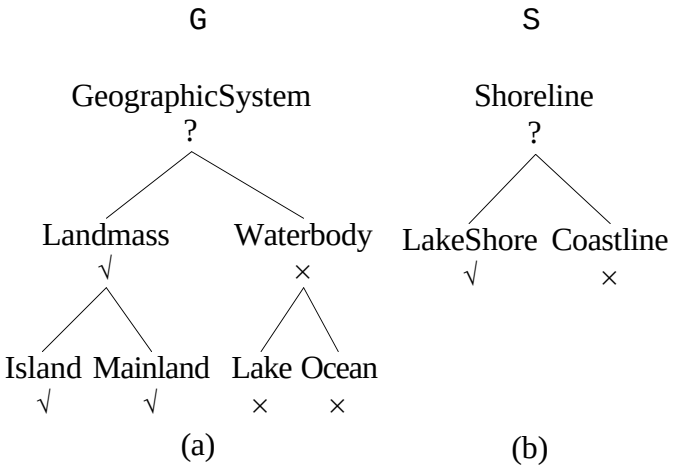

Figure 4. (a) Geo-system and (b) Shore specialization hierarchies

We formalize domain taxonomies as in (Mackworth et al., 1985). Assume that the size of each variable domain is a power of two which is structured into a complete binary tree of height m. That is, $D_X = \{a_i \mid 1 \le i \le 2^m\}$ and domains in the tree for D_X are D_X^{ks} ($0 \le k \le m$, $1 \le s \le 2^k$) where the pair (k,s) specifies the node in the tree. The integer k is the distance from the root and the integer s is the number of the node at distance k from the root counting from the left starting at 1. The root domain, D_X^{01}, is D_X. For $0 \le k < m$, the children of (k,s) are $(k+1,2s\text{-}1)$ and $(k+1,2s)$ with the conditions that:

[6] $$D_X^{ks} = D_X^{(k+1)(2s-1)} \cup D_X^{(k+1)2s} \text{ and}$$

[7] $$D_X^{(k+1)(2s-1)} \cap D_X^{(k+1)2s} = \emptyset.$$

[12]It should be noted that Mackworth, Mulder and Havens (1985) describes HAC in a different way. This representation makes it easier to exploit order on sets of real numbers.

These two conditions ensure respectively that the children cover their parent exhaustively and mutually exclusively. The leaf domains are $D_{\mathsf{X}}^{mi} = \{a_i\}$ $(1 \le i \le 2^m)$. Thus for Figure 4:

D_{G}^{21} = Island = {*island*},

D_{G}^{22} = Mainland = {*mainland*},

D_{G}^{23} = Lake = {*lake*},

D_{G}^{24} = Ocean = {*ocean*},

D_{G}^{11} = Landmass,

D_{G}^{12} = Waterbody, and

D_{G}^{01} = GeographicSystem.

For a variable X, the relationship between Δ_{X} and the nodes in the tree for D_{X} is defined by the marks M_{X}^{ks} on nodes (k,s) for $0 \le k \le m$ and $1 \le s \le 2^k$. The interpretation for these marks is

[8] $$M_{\mathsf{X}}^{ks} = \begin{cases} \surd & \text{if } D_{\mathsf{X}}^{ks} \subseteq \Delta_{\mathsf{X}} \\ ? & \text{if } D_{\mathsf{X}}^{ks} \not\subseteq \Delta_{\mathsf{X}} \text{ and } D_{\mathsf{X}}^{ks} \cap \Delta_{\mathsf{X}} \neq \varnothing \\ \times & \text{if } D_{\mathsf{X}}^{ks} \cap \Delta_{\mathsf{X}} = \varnothing \end{cases}.$$

The dynamic domain, Δ_{X}, is the union of the domains of all nodes marked '√':

[9] $$\Delta_{\mathsf{X}} = \bigcup \{ D_{\mathsf{X}}^{ks} \mid M_{\mathsf{X}}^{ks} = \surd \}.$$

However, some of the nodes marked '√' are redundant since all descendants of nodes marked '√' are also marked '√' and all descendants of nodes marked '×' are also marked '×'. These two observations are central to the HACR method. The domain taxonomy permits consistency algorithms to retain or eliminate whole subtrees as a unit, simply by manipulating the marks. We introduce the following notation for non-redundant node domains. $\Delta_{\mathsf{X}}^{\surd}$ is the smallest set of domains in the tree for D_{X} such that $\bigcup \Delta_{\mathsf{X}}^{\surd} = \Delta_{\mathsf{X}}$.

Similarly, $\Delta_X^{\times}$ is the smallest set of domains in the tree for D_X such that $\cup\Delta_X^{\times} = D_X \setminus \Delta_X$.

For instance, in Figure 4, $\Delta_X^{\surd}$ = {Landmass} and $\Delta_S^{\times}$ = {Coastline}.

To delete inconsistent values from Δ_X a consistency algorithm only needs to change marks in subtrees rooted at nodes with domains in $\Delta_X^{\surd}$ and possibly marks on the path back to the root. Similarly, to add values to Δ_X, only marks on nodes in paths from the root to nodes with domains in $\Delta_X^{\times}$ and marks on nodes in subtrees rooted at nodes with domains in $\Delta_X^{\times}$ need to be changed. The tree itself is an efficient representation for $\Delta_X^{\surd}$ and $\Delta_X^{\times}$ because they can be generated by a simple depth-first search of the subtree with nodes marked '?'. The ReviseHACR algorithm described in section 4.3 makes extensive use of these properties.

We extend hierarchical domains for real intervals as follows. The domain of each node in a taxonomy represents a real interval. Thus, instead of symbols, nodes are associated with the lower and upper bounds of the intervals they represent. Conceptually, these trees are infinite but they can be represented finitely by terminating branches with nodes whose domains are elements of $\Delta_X^{\surd}$ and $\Delta_X^{\times}$.

For example, consider the previous Echidna scheduling program of Figure 2 and the goal:

```
[10] ?- S ∈ [0, 4],
        schedule([task(0,1), task(2.75,1), task(S,0.875)], task(0,4.875)).
```

Figure 5 illustrates the scheduling problem schematically. Each solid arrow in figure 5 represents a task with the start time at the tail and the duration in the middle. Two tasks of one time unit in duration are already placed in the super task starting at 0 with duration 4.875, and a third task starting at time S and with duration 0.875 must be scheduled. The two dotted lines point to the arc-consistent intervals for S.

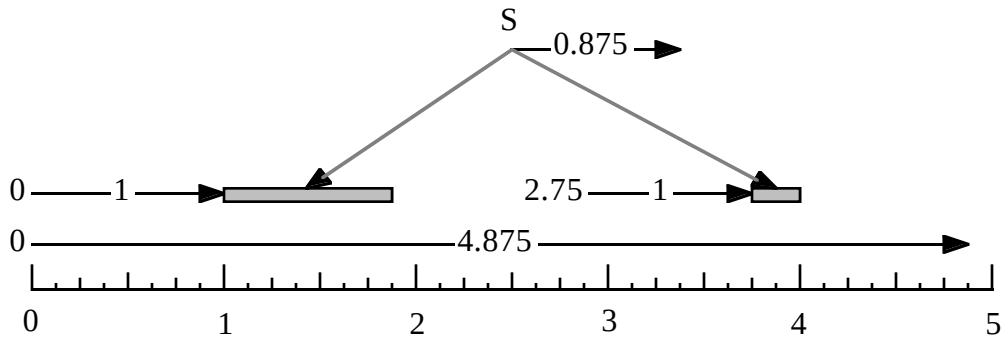

Figure 5 A scheduling problem

When the CSP induced by this goal is made arc consistent, $\Delta_S = [1, 1.875] \cup [3.75, 4]$, as shown by the shaded rectangles in figure 5. HACR represents Δ_S using the structure shown in figure 6. The root domain is D_S and the domains for the two children of each node are roughly the lower and upper halves of their parent domain. The relationship between a parent and its two children is

[11]
$$[x, y] \quad \Rightarrow \quad [x, mid(x,y)) \quad [mid(x,y), y]$$

where $x < mid(x,y) < y$. The types of interval bounds (open or closed) associated with x and y in the children are inherited from the parent and one of the bounds associated with $mid(x,y)$ is open while the other is closed. There are several reasonable definitions for $mid(x,y)$. If an unbounded precision (eg. rational) number system is used, then the mean $((x+y)/2)$ or the mediant[13] (Graham et al., 1989) can be used. If a floating point number system is used, then the number nearest to the mean or the median number in the system between x and y can be used. Cleary (1987) studies the efficiency of these two options for the floating point number system.

[13]The *mediant* of two rational numbers p/q and r/s is $(p+r)/(q+s)$.

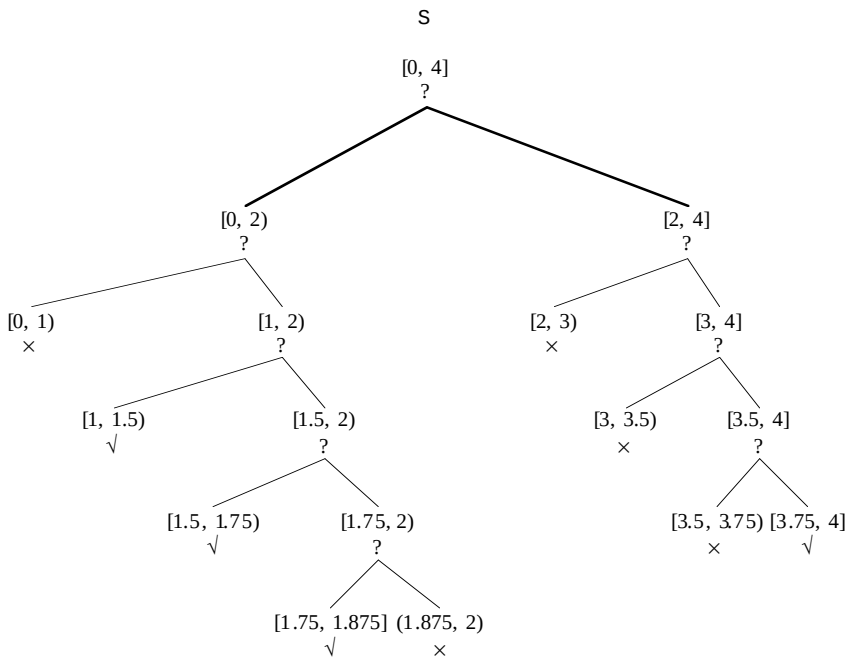

Figure 6. The domain representation for a real variable S.

Echidna presently represents interval bounds using 64-bit floating point numbers and defines *mid*(*x*,*y*) to be the floating point number nearest to the mean of *x* and *y*. Future versions may allow the programmer to specify both the number system and the definition of *mid*(*x*,*y*). Currently, all bounds are closed so that they need only be stored explicitly for the root interval. The bounds for any other node is calculated via the path from the node to the root. The fact that both children of a node with interval [*x*, *y*] contain *mid*(*x*, *y*) can be accommodated by our theory by complicating the meaning of node marks.

4.3 ReviseHACR

HAC and HACR are quite similar algorithms. Internally, their respective arc revision procedures, ReviseHAC and ReviseHACR are also similar. ReviseHAC relies on precompiled extensional constraints. It can be generalized for constraints on any number of variables, but for simplicity we describe it only for binary constraints. Assume that there is a single source variable S for all hyperarcs (T, C). Constraints are compiled into predicates which can be used to update the marks in the domain taxonomies using only the symbols which label their nodes. These predicates test if *all* or *some* value(s) in a subset D_{T}^{ks} of D_{T} are consistent with some value in a subset D_{S}^{lr} of D_{S}. Conceptually, both ReviseHAC(T, C) and ReviseHACR(T, C) perform the assignment of a new mark M_{T}^{ks} to one of its three possible values according to:

[12] $$M_{\mathsf{T}}^{ks} \leftarrow \begin{cases} \surd & \text{if } D_{\mathsf{T}}^{ks} \subseteq \pi_{\mathsf{T}}(\mathsf{C}) \\ ? & \text{if } D_{\mathsf{T}}^{ks} \not\subseteq \pi_{\mathsf{T}}(\mathsf{C}) \text{ and } D_{\mathsf{T}}^{ks} \cap \pi_{\mathsf{T}}(\mathsf{C}) \neq \emptyset \\ \times & \text{if } D_{\mathsf{T}}^{ks} \cap \pi_{\mathsf{T}}(\mathsf{C}) = \emptyset \end{cases}$$

for each subset D_{T}^{ks} of D_{T}. If all values in D_{T}^{ks} are consistent with some value in some $D_{\mathsf{S}}^{lr} \in \Delta_{\mathsf{S}}^{\surd}$, then M_{T}^{ks} remains '√'. Otherwise if some values are consistent with some value in some $D_{\mathsf{S}}^{lr} \in \Delta_{\mathsf{S}}^{\surd}$, then M_{T}^{ks} is changed to '?' and the children domains, $D_{\mathsf{T}}^{(k+1)(2s-1)}$ and $D_{\mathsf{T}}^{(k+1)2s}$, are considered[14]. Otherwise none of the values are consistent so M_{T}^{ks} is set to '×'. By repeating this procedure for every node in $\Delta_{\mathsf{T}}^{\surd}$, the new domain Δ_{T} is constructed according to the assignment $\Delta_{\mathsf{T}} \leftarrow \pi_{\mathsf{T}}(\mathsf{C})$. Mackworth et al. (1985) show that ReviseHAC is a full arc revision algorithm.

For our algorithm, ReviseHACR, domains are infinite non-discrete sets, so precompiling predicates is impossible. Instead ReviseHACR(T, C) computes $\pi_{\mathsf{T}}(\mathsf{C})$ by generating a set of intervals whose union is $\pi_{\mathsf{T}}(\mathsf{C})$[15]. The intervals are generated one at a time and the new Δ_{T} is accumulated from them. In our development, let C be an n-ary constraint and $\nu(\mathsf{C}) = \{\mathsf{S}_1, \ldots, \mathsf{S}_{n-1}, \mathsf{S}_n = \mathsf{T}\}$. Iterating through $\Delta_{\mathsf{T}}^{\surd}$ and searching for tuples $(D_{\mathsf{S}_1}^{l_1 r_1}, \ldots, D_{\mathsf{S}_{n-1}}^{l_{n-1} r_{n-1}}) \in \Delta_{\mathsf{S}_1}^{\surd} \times \ldots \times \Delta_{\mathsf{S}_{n-1}}^{\surd}$ which are consistent with C will be very inefficient as *n*

---

[14]Notice that for $D_{\mathsf{T}}^{ks} \in \Delta_{\mathsf{T}}^{\surd}$, changing M_{T}^{ks} from '√' to '?' implicitly removes D_{T}^{ks} from $\Delta_{\mathsf{T}}^{\surd}$ and adds its children.

[15]Actually, when floating point numbers are used, the generated set of intervals may be a superset of $\pi_{\mathsf{T}}(\mathsf{C})$. This is because bounds outside the floating point number system must be rounded to floating point numbers. As long as bounds are rounded outwardly, so the floating point interval is a superset of the actual interval, ReviseHACR(T, C) will delete only inconsistent values from the domain of T.

increases. Instead, ReviseHACR updates Δ_T by computing $\pi_T(C)$ from $\{\Delta^{\surd}_{S_1},\ldots,$ $\Delta^{\surd}_{S_{n-1}}\}$.

ReviseHACR would also be a full arc revision algorithm if the marks could be set exactly as specified in formula [12] above. However, for some CSPs it is possible that an unbounded amount of refinement will be required. This is not a problem with ReviseHAC because it manipulates finite domains structured as finite taxonomies. Thus ReviseHAC is guaranteed to terminate. For infinite real taxonomies (and in fact any dense ordered set), a full arc revision algorithm would not be expected to terminate. To avoid this eventuality, Echidna attaches a positive integer *precision* `P` to each variable `X` in the list `Vars` using the built-in predicate, `precision(Vars, P)`. `P` is the maximum distance from the root to any node in the taxonomy for Δ_X. When ReviseHACR determines that D^{ks}_X should refined, that is M^{ks}_X should be set to '?' and its children analyzed, if the node (k, s) is at the precision limit ($k = P$) then M^{ks}_X is left '√'. For this reason, ReviseHACR is only a partial arc revision algorithm but it can approximate a full arc revision algorithm by increasing the precision of variables as necessary.[16]

We return to this issue of computing the new domain Δ for Δ_T such that:

[13] $$\pi_T(C) \subseteq \Delta \subseteq \Delta_T.$$

We implement this specification by computing the set Δ which is as close to $\pi_T(C)$ as possible given the current precision of `T`. This is done using a set of additional "temporary" marks associated with the nodes of the taxonomy for D_T:

[14] $$\{TM^{ks}_T \mid k \geq 0,\ 1 \leq s \leq 2^k\}.$$

These temporary marks represent Δ in the same way that the set $\{M^{ks}_T\}$ represents Δ_T. That is:

[15] $$\Delta = \bigcup\{D^{ks}_T \mid TM^{ks}_T = \surd\}.$$

[16] Precision can be increased during execution under program control.

We define $\Delta^{\surd}$ as the smallest subset of $\{D_{\mathsf{T}}^{ks} \mid TM_{\mathsf{T}}^{ks} = \surd\}$ such that $\Delta = \cup\Delta^{\surd}$. Likewise, $\Delta^{\times}$ is the smallest subset of $\{D_{\mathsf{T}}^{ks} \mid TM_{\mathsf{T}}^{ks} = \times\}$ such that $D_{\mathsf{T}} \backslash \Delta = \cup\Delta^{\times}$.

The full procedure ReviseHACR is shown in Figure 7. Its principle subprocedure is MarkTemp(D_{T}^{01}, I) which adds an approximation of the interval I to Δ by searching through the taxonomy of the variable T starting at the root interval D_{T}^{01}. That is, given $\Delta = S$, MarkTemp(D_{T}^{01}, I) updates Δ so $\Delta = S \cup approx(I, P)$ where $approx(I, P)$ is the smallest superset of I which can be represented in the taxonomy for T at its current precision P. An approximation of $\pi_{\mathsf{T}}(\mathsf{C})$ accumulated in Δ by repeatedly calling MarkTemp with a set of intervals whose union is $\pi_{\mathsf{T}}(\mathsf{C})$. MarkTemp does not require that the set of intervals is disjoint.

The ReviseHACR algorithm operates as follows. In line 3, all temporary marks are set to '×' effectively making Δ empty[17]. In line 4, the algorithm sets Δ to $approx(\pi_{\mathsf{T}}(\mathsf{C}), P)$ using MarkTemp, as discussed above. In line 5, the set of marks, Less, are collected which need to changed in updating Δ_{T} to be the intersection of its old value and Δ. In line 6, Δ_{T} is updated appropriately. Less is the set of node domains with fewer values in the new value for Δ_{T}. A node domain has fewer values if its mark is changed to a smaller value according to the order $\times < ? < \surd$. The operation specified by lines 5-6 can be implemented by a constrained depth first search of the taxonomy for T which has a form similar to MarkTemp, which is described next. Since Less is empty if and only if the Δ_{T} is not changed, ReviseHACR returns true at line 7 if and only if some inconsistent values are deleted from Δ_{T}.

The subprocedure, MarkTemp(D_{T}^{ks}, I), is now described in more detail. It considers three cases (in lines 11, 12 and 13 respectively). In the first case, it returns without changing any temporary marks (in line 11) if any of the following conditions are true. If $M_{\mathsf{T}}^{ks} = \times$, then $D_{\mathsf{T}}^{ks} \cap \Delta_{\mathsf{T}} = \emptyset$ meaning that D_{T}^{ks} has already been removed from Δ_{T}. If $TM_{\mathsf{T}}^{ks} = \surd$, then

---

[17]Actually, the temporary marks can be set to '×' on an as needed basis by initially only setting TM_{T}^{01} to '×'. Care must then be taken later in MarkTemp to set temporary marks of children of nodes with intervals in $\Delta^{\times}$ to '×' if they are ever accessed.

$D_{\mathsf{T}}^{ks} \subseteq \Delta$ meaning that an interval covering D_{T}^{ks} has already been generated. If $I \cap D_{\mathsf{T}}^{ks} = \emptyset$ then the I misses D_{T}^{ks} and the subtree rooted at (k, s) can be left as is.

Otherwise, in the second case (in line 12), if $D_{\mathsf{T}}^{ks} \subseteq I$ (indicating that I covers D_{T}^{ks}) or k = P (indicating that the precision limit of T has been reached), then D_{T}^{ks} is added to Δ by assigning $TM_{\mathsf{T}}^{ks} = \surd$. At the precision limit, some values outside I may be added to Δ, but only ones which require greater precision to eliminate. Note that $I \cap D_{\mathsf{T}}^{ks} = \emptyset$ and $D_{\mathsf{T}}^{ks} \subseteq I$ can be tested efficiently by comparing appropriate bounds.

```
1   procedure ReviseHACR(T, C):
2   begin                                                    {Δ_T = S}
3       for 0≤k do for 1≤s≤2^k do TM_T^ks ← ×;               {Δ = Ø}
4       for I ⊆ π_T(C) do MarkTemp(D_T^01, I);               {Δ = approx(π_T(C), P_T)}
5       Less ← {D_T^ks | TM_T^ks < M_T^ks};                  (* × < ? < √ *)
6       for D_T^ks ∈ Less do M_T^ks ← TM_T^ks;               {Δ_T = S ∩ Δ}
7       return Less ≠ Ø
8   end;

9   procedure MarkTemp(D_T^ks, I):
10  begin
11      if (M_T^ks = ×) ∨ (TM_T^ks = √) ∨ (I ∩ D_T^ks = Ø) then return
12      else if (D_T^ks ⊆ I) ∨ (k = P_T) then TM_T^ks ← √
13      else begin                                           {I∩D_T^ks≠Ø ∧ D_T^ks⊈I ∧ TM_T^ks≠√}
14          TM_T^ks ← ?;
15          MarkTemp(D_T^(k+1)(2s-1), I); MarkTemp(D_T^(k+1)2s, I);
16      end
17  end;
```

Figure 7. ReviseHACR: a revision algorithm for hierarchical numeric domains

Otherwise, in the third case (in line 13), since the branches at lines 11 and 12 were not taken, $I \cap D_T^{ks} \neq \emptyset$, $D_T^{ks} \not\subseteq I$, and $TM_T^{ks} \neq \surd$. Thus, TM_T^{ks} must be set to '?' and its children must be analyzed, as they are in lines 14 and 15 respectively[18].

[18]There is a case where MarkTemp leaves (TM_T^{ks} = ?) when the children of node (k, s) are both temporarily marked '√'. If k+1 is the precision limit and the point on the boundary between the two children is in I, the two recursive calls on line 15 will temporarily mark both children '√'. In this case TM_T^{ks} should not be '?' because every value in D_T^{ks} is still in the temporary domain Δ because of the two children. This complication does not affect the correctness of the algorithm. However, it can be detected after line 15 by testing if (k+1 = P_T) ∧ ($TM_T^{(k+1)(2s-1)}$ = √) ∧ ($TM_T^{(k+1)2s}$ = √) is true and it can be corrected by

4.4 Computing Projections

Let C be an equality or inequality with $\nu(C) = \{S_1, \ldots, S_{n-1}, S_n = T\}$. Computing projections is facilitated by transforming the formula for C into an equivalent formula for the constraint *iso*(T, C) which isolates the variable T. Thus, C and *iso*(T, C) contain the same set of mappings. For instance, *iso*(X, X·Y=Z) = 'X=Z÷Y'. The constraint *iso*(T, C) is of the form 'T *r* E' where $r \in \{=, <, \leq, >, \geq\}$, E is a numeric expression, and $T \notin \nu(E) = \{S_1, \ldots, S_{n-1}\}$. It is convenient to use $f_E{:}\Delta_{S_1}\times\ldots\times\Delta_{S_{n-1}} \rightarrow \mathbb{R}$ to denote the function of $(S_1, \ldots, S_{n-1})$ defined by E and the current dynamic domains of $S_1, \ldots, S_{n-1}$. The range of f_E is

[16] $$range(f_E) = \{f_E(a_1,\ldots,a_{n-1}) \mid (\exists(a_1,\ldots,a_{n-1}) \in \Delta_{S_1}\times\ldots\times\Delta_{S_{n-1}})\}.$$

The projection $\pi_T(C)$ can now be computed from the variable T and the numeric expression E. Given the ability to isolate variables, Sections 4.4.1 and 4.4.2 describe the computation of projections of equalities and inequalities, respectively. Section 4.4.3 uses the results for inequalities to compute projections for disjunctive inequalities. But first, two restrictions are made on domains and constraints to shorten this presentation. These two restrictions are also used to reduce the complexity of our implementation of Echidna. They are as follows:

1. All intervals in domain taxonomies are of the form $[x_1, x_2]$ where both the lower and upper bounds are closed and all inequalities are the nonstrict type (ie- $\leq$ and $\geq$).

2. All equalities contain at most one function symbol and all inequalities contain no function symbols. Consequently, constraints are of the form '$A_1=A_2$', '$A_1\leq A_2$', '$A_1+A_2=A_3$', '$A_1\cdot A_2=A_3$', '$A_1^{A_2}=A_3$', '*sin*$(A_1)=A_2$', '$A_1\leq A_2 \vee A_1\geq A_3$', *et cetera* where A_1, A_2, and A_3 are either real variables or real constants.

Cleary (1987) describes some of these issues involved in removing the first restriction. The second restriction makes computing *iso*(T, C) trivial for constraints involving only invertible functions. A full presentation of how to compute projections of constraints involving more functions with open and closed intervals is in preparation (Sidebottom, 1992). We consider here only computing $\pi_T(C)$ for an arbitrary constraint C subject to these two restrictions.

Echidna satisfies restriction 2 by introducing intermediate variables to decompose complex constraints into an equivalent simpler set . For instance, the `onCircle/2` predicate of [1] is transformed to:

---

temporarily marking node (*k*, *s*) and its ancestors '√' whenever both their children are temporarily marked '√'.

```
onCircle(p(X,Y), c(p(A,B), R)) :-
   R > 0,
   T1 = X - A,
   T2 = T1·T1,      % T2 = (X - A)^2
   T3 = Y - B,
   T4 = T3·T3,      % T4 = (Y - B)^2
   T5 = R·R,        % T5 = R^2
   T2 + T4 = T5.    % (X - A)^2 + (Y - B)^2 = R^2
```

where T*i* are new intermediate variables ($1 \le i \le 5$). The domain D_{Ti} of intermediate variable T*i* in a constraint T*i* = E is [*min range*(f_E), *max range*(f_E)]. All subexpressions of real constraints are decomposed in this same way. The domains of intermediate variables can be calculated efficiently because f_E is either: 1) a constant; 2) a variable; or 3) a numeric function applied to variables and constants. In the first case, $f_E = a$ where *a* is a real constant then *min range*(f_E) = *max range*(f_E) = *a*. In the second case, $f_E(X) = X$ giving *min range*(f_E) = *min* Δ_X and *max range*(f_E) = *max* Δ_X. We order the children of each node in the taxonomy with the domains containing smaller values to the left and the domains containing larger values to the right. Consequently, *min* Δ_X and *max* Δ_X are in the leftmost and rightmost domains in $\Delta_X^{\surd}$, respectively:

[18] *min* $\Delta_X = a_1$ where $[a_1,a_2] = D_X^{ks} \in \Delta_X^{\surd}$ is such that $s = min\{s' \mid D_X^{k's'} \in \Delta_X^{\surd}\}$

[19] *max* $\Delta_X = a_2$ where $[a_1,a_2] = D_X^{ks} \in \Delta_X^{\surd}$ is such that $s = max\{s' \mid D_X^{k's'} \in \Delta_X^{\surd}\}$.

The leftmost node domain in [18] can be found by following the path of left descendents from the root node until a node not marked '?' is found. If the node is marked '√' then its lower bound is *min* Δ_X. Otherwise, the lower bound of its sibling is *min*Δ_X. Similarly, *max* Δ_X can be found by following the path of right descendents from the root.

In the last case, E involves a numeric function. The bounds, *min range*(f_E) and *max range*(f_E), can be calculated from the respective minima and maxima of the function arguments by analyzing the monotonicity and continuity properties of the function (Bundy, 1984; Ratschek and Rokne, 1984). This analysis is applied in Echidna for the arithmetic, exponential, logarithmic, root extraction, and trigonometric functions.

4.4.1 Equalities

For simple equalities of the form *iso*(T, C) = 'T = E', the projection $\pi_T(C)$ = *range*(f_E). If E is the constant *a* then *range*(f_E) = {*a*}. Otherwise,

[20] *range*(f_E) =

$$\bigcup\{f_{\mathsf{E}}(D_{\mathsf{S}_1}^{k_1 s_1},\ldots,D_{\mathsf{S}_{n-1}}^{k_{n-1} s_{n-1}}) \mid (D_{\mathsf{S}_1}^{k_1 s_1},\ldots,D_{\mathsf{S}_{n-1}}^{k_{n-1} s_{n-1}}) \in \Delta_{\mathsf{S}_1}^{\surd} \times \ldots \times \Delta_{\mathsf{S}_{n-1}}^{\surd}\}.$$

The Region Splitting theorem of Bundy (1984) ensures the correctness of this approach for computing $\pi_{\mathsf{T}}(\mathsf{C})$. Bundy gives a general theory of functions applied to intervals whereas Alefeld and Herzberger (1983) give some specific results for the arithmetic functions. The following formulas from Alefeld and Herzberger specify the four arithmetic operations on intervals:

[22] $[x_1, x_2] + [y_1, y_2] = [x_1+y_1, x_2+y_2]$

[23] $[x_1, x_2] - [y_1, y_2] = [x_1-y_2, x_2-y_1]$

[24] $[x_1, x_2]\cdot[y_1, y_2] = [min\{x_1\cdot y_1,x_1\cdot y_2,x_2\cdot y_1,x_2\cdot y_2\}, max\{x_1\cdot y_1,x_1\cdot y_2,x_2\cdot y_1,x_2\cdot y_2\}]$

[25] $[x_1, x_2] \div [y_1, y_2] = [x_1, x_2]\cdot[1/y_2, 1/y_1]$ $\qquad (0 \notin [y_1, y_2])$

We generalize [25] by considering divisors which include zero in their domains. In this case, the quotient is the union of two disjoint intervals. For instance,

[26] $[1, 1] \div [-2, 3] = (-\infty,-1/2] \cup [1/3,+\infty)$.

This is accommodated by splitting the denominator at zero:

[27] $[1,1] \div [-2,3] = [1,1] \div ([-2,0] \cup [0,3])$,

and appealing to the Region Splitting theorem which yields:

[28] $[1,1] \div ([-2,0] \cup [0,3]) = ([1,1] \div [-2,0]) \cup ([1,1] \div [0,3])$.

These two disjoint expressions can then be evaluated using [28] by replacing forms such as $1\div0$ by the limit as the denominator approaches zero from within its interval. For example, $[1,1] \div [-2,0] = [1\div(0–), 1\div-2]$ where $1\div(0–) = -\infty$, the limit of '$1\div x$' as x approaches zero from below. Likewise, the exponential, logarithmic, root extraction, and trigonometric functions can be handled similarly by analyzing periodicity and monotonicity properties and by taking limits at points of discontinuity.

The number of expressions of the form $f_{\mathsf{E}}(D_{\mathsf{S}_1}^{k_1 s_1},\ldots,D_{\mathsf{S}_{n-1}}^{k_{n-1} s_{n-1}})$ evaluated in [20] above can be reduced by combining adjacent intervals in $\Delta_{\mathsf{S}_i}^{\surd}$ $(1 \le i < n)$. For instance, in the scheduling example of Figure 6, $\Delta_{\mathsf{S}}^{\surd} = \{[1,1.5), [1.5,1.75), [1.75,1.875), [3.75,4)\}$ but after combining adjacent intervals, only the set $\{[1,1.875), [3.75,4)\}$ need be considered to calculate $\pi_{\mathsf{T}}(\mathsf{C})$ where C is an equality with $\mathsf{S} \in \nu(\mathsf{C})$. The complexity of computing $\pi_{\mathsf{T}}(\mathsf{C})$ is another reason for decomposing equalities as described above. Even after

adjacent intervals are combined, $|\Delta^{\surd}_{S_1} \times \ldots \times \Delta^{\surd}_{S_{n-1}}|$ increases rapidly with *n*.. After constraints are decomposed, C is transformed into a set of constraints with arity not greater than three. The size of each $\Delta^{\surd}_{S_i}$ is further limited from below by the setting of the precision for each variable (as previously described).

4.4.2 Inequalities

For inequalities of the form *iso*(T, C) = 'T $\leq$ E', the projection $\pi_T(C)$ should range between *min* Δ_T and the *max range*(f_E). Precisely stated,

[29] $$\pi_T(C) = [min\ \Delta_T, max\ range(f_E)].$$

Similarly, if *iso*(T, C) = 'T $\geq$ E' then

[30] $$\pi_T(C) = [min\ range(f_E), max\ \Delta_T].$$

Since f_E is a constant or a variable by restriction 2, *min* Δ_X, *max* Δ_X, *min range*(f_E), and *max range*(f_E) can be calculated using [17-19] above.

4.4.3 Disjunctive Inequalities

If C is a disjunctive inequality of the form $C_1 \vee C_2$ then the projection $\pi_T(C)$ depends on whether T appears in one or both of $v(C_1)$ and $v(C_2)$. If both $T \in v(C_1)$ and $T \in v(C_2)$ then values which satisfy[19] either disjunct can be used to satisfy the whole constraint. The projection is:

[31] $$\pi_T(C) = \pi_T(C_1) \cup \pi_T(C_2).$$

For the other case, T only appears in one expression. Assume that $T \in v(C_1)$ and $T \notin v(C_2)$ (the other case is symmetrical). Then $\pi_T(C)$ depends on the satisfiability of C_2. If C_2 = '$E_1 \leq E_2$' then C_2 can be efficiently tested for satisfiability given *min range*(f_{E_1}) and *max range*(f_{E_2}) which in turn can be computed using [17-19]. It is possible to show that C_2 is satisfiable if and only if

[32] $$min\ range(f_{E_2}) \leq max\ range(f_{E_2}).$$

Similarly, if C_2 = '$E_1 \geq E_2$' then C_2 satisfiable if and only if

[33] $$max\ range(f_{E_2}) \geq min\ range(f_{E_2}).$$

---

[19]By satisfiability here, we mean local satisfiability of the constraint under consideration.

If C_2 is satisfiable, then there exists $\mu' \in C_2$ which can be extended arbitrarily to a mapping $\mu \in C$. Specifically, if $\nu(C_1)\backslash\nu(C_2) = \{X_1,\ldots,X_m\}$ and $(a_1,\ldots,a_m) \in \Delta_{X_1}\times\ldots\times\Delta_{X_m}$ then $\mu = \mu' \cup \{(X_1,a_1),\ldots,(X_m,a_m)\} \in C$. Since $T \in \nu(C_1)\backslash\nu(C_2)$, $\pi_T(C) = \Delta_T$ if C_2 is satisfiable. If C_2 is unsatisfiable, then all $\mu \in C$, when restricted to $\nu(C_1)$, must satisfy C_1. Thus, $\pi_T(C) = \pi_T(C_1)$. To summarize,

$$[34]\ \pi_T(C_1 \vee C_2) = \begin{cases} \pi_T(C_1) \cup \pi_T(C_2) & \text{if } T \in \nu(C_1) \wedge T \in \nu(C_2) \\ \Delta_T & \text{if } T \in \nu(C_1) \wedge T \notin \nu(C_2) \wedge C_2 \text{ is satisfiable} \\ \Delta_T & \text{if } T \notin \nu(C_1) \wedge T \in \nu(C_2) \wedge C_1 \text{ is satisfiable} \\ \pi_T(C_1) & \text{if } T \in \nu(C_1) \wedge T \notin \nu(C_2) \wedge C_2 \text{ is unsatisfiable} \\ \pi_T(C_2) & \text{if } T \notin \nu(C_1) \wedge T \in \nu(C_2) \wedge C_1 \text{ is unsatisfiable.} \end{cases}$$

5. Examples and Comparisons

This section provides some comparisons of Echidna's real number capabilities with other major CLP systems on example problems. These examples were run using Echidna version 1.0 on a Sun UNIX Sparcstation. The number representation is 64-bit floating point numbers. A linear technique is used for domain splitting.

5.1 Finding Zeros of Functions

Echidna can generate numerical solutions to many equations with varying precision using the built-in predicate `precision`. For instance, consider the following query. We employ the predicate `precision([X],8)` to limit initially the precision of the variable `X` to 8-bits. The predicate `split([X])` is used to invoke a case analysis search for solutions for `X`.

```
[35] ?- X ∈ [-1000, 1000),
        precision([X], 8),
        (X - 1)·(X - 2) = 0,
        split([X]).
```

Since Echidna's numeric constraint solving system is incomplete, it outputs answers which approximate the solutions in terms of domains for variables[20]. All solutions to the CSP are contained in the answers although every answer does not necessarily contain a solution. For query [35], Echidna computes the following answer (containing the two solutions, `X = 1` and `X = 2`):

```
   X ∈ [0, 7.8125]       ;
   no.
```

[20]Recall that *answers* are what CLP systems compute and *solutions* are substitutions which make the query a logical consequence of the program.

More precise approximations of the solution can be obtained by computing answers with smaller domains. This can be achieved by increasing the precision of X. For instance, if the precision is set to 16, the following answers are computed:

```
X ∈ [0.9765625, 1.00708] ;
X ∈ [1.983643, 2.01416]  ;
no.
```

As the precision is further increased, more false answers are excluded. Setting the precision to 32 produces the following approximate solutions:

```
X ∈ [0.9999997, 1.0000002] ; % solution here
X ∈ [1.9999998, 2.0000003] ; % solution here
no.
```

For an example using trigonometric functions, consider the following query[21]:

```
[36] ?- X ∈ [0.001, 1],
        precision([X], 8),
        sin(1/X) = 0,
        split([X]).
```

For reference, a plot of $sin(1/x)$ for $x \in [0, 1]$ is given in figure 8. For this query, Echidna answers with:

```
X ∈ [0.001, 0.004902344]         ; % many solutions here
X ∈ [0.004902344, 0.008804688]   ; % many solutions here
…
X ∈ [0.03612109, 0.04002344]     ; % some solutions here
X ∈ [0.04392578, 0.04782813]     ; % one solution here
X ∈ [0.05173047, 0.05563281]     ; % one solution here
X ∈ [0.0634375, 0.06733984]      ; % one solution here
X ∈ [0.07904688, 0.08294922]     ; % one solution here
X ∈ [0.1024609, 0.1063633]       ; % one solution here
X ∈ [0.1570938, 0.1609961]       ; % one solution here
X ∈ [0.3170898, 0.3209922]       ; % one solution here
no.
```

The first several answers contain many solutions and cannot be broken down any further because of the precision limit. The last seven answers contain one solution each. Increasing the precision results in more answers which contain single solutions.

[21] We use 0.001 as the lower bound for X because the special techniques for dealing with zeros in denominators and infinite bounds have not yet been implemented in Echidna. 0.001 can be replaced by any number arbitrarily close to 0 which can be represented as a floating point number.

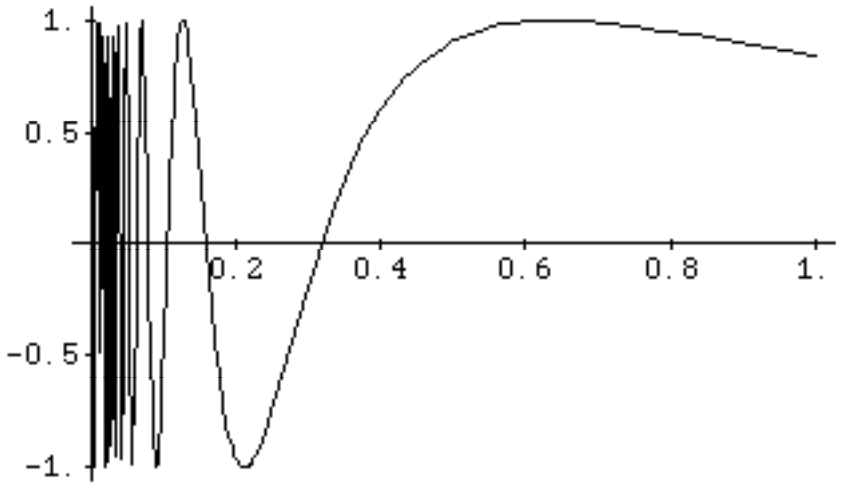

Figure 8. A plot of $sin(1/x)$ for $x \in [0, 1]$

Systems like CLP(ℝ) (Jaffar & Michaylov, 1987), Prolog-III (Colmerauer, 1990), and CAL (Aiba et al., 1988) are based on symbolic manipulation of constraints. Their solutions consists of a set of constraints in some solved form. Since CLP(ℝ) and Prolog-III can solve only linear constraints, they cannot solve the above queries. CAL is powerful enough to find the two solutions to the polynomial query, but cannot solve the trigonometric query.

For real number constraint processing, Echidna is most similar to BNR Prolog (Older & Vellino, 1990). They both use arc consistency algorithms to remove values from the dynamic domains of variables. Both languages can find zeros of many functions numerically to reasonable accuracy efficiently. BNR Prolog provides primitives for programming case analysis algorithms, like `split`, which compute solutions to varying accuracy, but it has no programmable control over the accuracy of its consistency algorithms. The Echidna predicate, `precision`, provides this necessary control. We are currently investigating more programmable ways of implementing case analysis algorithms in a control meta-language.

5.2 Geometry

The query [2] in section 2, which uses the `onCircle` predicate predicate defined by [1], can be augmented with precision and case analysis calls. The resulting query is:

```
[37] ?- precision([A,B,R], 16),
        A ∈ [-100,100],
        B ∈ [-100,100],
        R ∈ [-100,100],
        C = c(p(A,B),R),
        onCircle(p(0,1), C),
        onCircle(p(1,0), C),
        onCircle(p(-1,0), C)
        split([A, B, R]).
```

Echidna finds the following answer which closely approximates the correct solution:

```
A ∈ [-0.003051758,0],
B ∈ [-0.003051758,0],
R ∈ [0.9979248, 1.000977]
```

Again, neither CLP(ℝ) nor Prolog-III can solve this query because they only process linear constraints while CAL can solve this query exactly.

5.3 Linear Equations

This example shows how the specialized linear constraint solving algorithms of CLP(ℝ) and Prolog-III are superior to both Echidna and BNR Prolog for linear constraints. Echidna, like BNR Prolog, can solve linear systems like:

```
[38] ?- precision([X,Y,Z], 16),
        X ∈ [-1000, 1000],
        Y ∈ [-1000, 1000],
        Z ∈ [-1000, 1000],
          X + 2·Y +   Z =  4,
        3·X +   Y + 5·Z =  9,
        7·X + 4·Y + 8·Z = 16,
        split([X, Y, Z]).
```

However, the time required increases rapidly with the number of variables and equations in the CSP. Arc consistency algorithms only consider constraints in isolation. Higher levels of consistency can be achieved using path consistency and *k*-consistency algorithms (Mackworth, 1977; Freuder, 1978), but these general algorithms cannot compete with specialized algorithms for linear constraints.

5.4 Scheduling

Given the scheduling program introduced earlier in Figure 2, consider the query:

```
[39] ?- precision([S1, S2], 3),
        S1 ∈ [0, 4],
        S2 ∈ [0, 4],
        schedule([task(S1, 2), task(S2, 1.5)], task(0, 4)).
```

The algorithms described here can compute the following answer[22]:

```
S1 ∈ [0, 0.5] ∪ [1.5, 2],
S2 ∈ [0, 0.5] ∪ [2, 2.5]
```

All other CLP systems known to the authors implement disjunctive constraints using logical disjunction in the underlying logic program. This is very inefficient since it introduces unnecessary nondeterminism in the program. For instance, the disjunctive inequality in the `noOverlap` predicate of Figure 2 could also be expressed as:

---

[22]Disjunctive inequalities are only partially implemented in the current version of Echidna.

```
S1 ≤ S2 - D1 ; S1 ≥ S2 + D2
```

using the disjunction connective (`;`) of Edinburgh syntax Prolog. This problematic version causes derivations to make an explicit choice about the relative order of the two tasks thereby introducing a backtrack point in the proof of calls to `noOverlap`. Hence the number of proofs for calls to `schedule` can be exponential in the number of tasks. Hierarchical domains directly support the implementation of disjunctive constraints. They make it possible to avoid this unnecessary inefficiency. It should be noted that Van Hentenryck and Deville (1991) introduce a new logical connective for CLP which can also be used to solve disjunctive scheduling problems more efficiently.

6. Conclusion

This paper has described how the CLP language Echidna augments a logic programming language with efficient algorithms for real number constraint processing. Echidna adds domain constraints, equalities, inequalities, and disjunctions of inequalities on real numeric expressions involving arithmetic, exponential, and trigonometric functions. The set of constraints supported by Echidna is richer than for the other numeric CLP languages cited.

Echidna's novel use of hierarchical domains and a hierarchical arc consistency algorithm makes it possible to process constraints with varying accuracy and to represent variable domains which are the union of disjoint sets of intervals. Initial experience programming in Echidna shows it solves many problems as effectively or more effectively than other CLP systems. We believe it is a suitable basis for the development of new CLP applications.

Acknowledgements

This work was supported by the Centre for Systems Science at Simon Fraser University, by the Science Council of British Columbia, and by the Natural Sciences and Engineering Research Council of Canada. We would also like to acknowledge the continuing efforts of the staff of the the Expert Systems Laboratory, including Miron Cuperman, Rod Davison, Russ Ovans and Sue Sidebottom, who have been instrumental in making the Echidna CLP system a reality.

References

Aiba, A., Sakai, K., Sato, Y. and Hawley, D. J. 1988. Constraint Logic Programming Language CAL. In Proc. The International Conference on Fifth Generation Systems. Ohmsha Publishers. Tokyo. 263-276.

Alefeld, G. and Herzberger, J. 1983. Introduction to Interval Computations. Academic Press, Toronto. 333 pages.

Allen, J. F. 1983. Maintaining Knowledge About Temporal Intervals. Communications of the ACM. 26 (11).

Buchberger, B. 1985. Grobner Bases: An Algorithmic Method in Polynomial Ideal Theory. In Multidimentional Systems Theory, Bose, N. K. (ed.).

Bundy, A. 1984. A Generalized Interval Package and Its Use for Semantic Checking. ACM Transactions on Mathematical Systems. 10 (4). 397-409.

Calvert, T., J. Dickinson, Dill, T., Havens W., Jones J. and L., B. 1991. An Intelligent Basis for Design. In Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing. Victoria, BC.

Cleary, J. G. 1987. Logical Arithmetic. Future Computing Systems. 2 (2). 125-149.

Colmerauer, A. 1990. An Introduction to Prolog III. Communications of the ACM. 33 (7). 69-90.

Davis, E. 1987. Constraint Propagation with Interval Labels. Artificial Intelligence. 32. 281-331.

Deville, Y. and Van Hentenryck, P. 1991. An Efficient Arc Consistency Algorithm for a Class of CSP Problems. In Proc. The International Joint Conference on Artificial Intelligence. Sydney, Australia. 325-330.

Freuder, E. C. 1978. Synthesizing Constraint Expressions. Communications of the ACM. 21 (11). 958-966.

Graham, R. L., Knuth, D. E. and Patashnik, O. 1989. Concrete Mathematics. Addison-Wesley, Don Mills, ON.

Havens, W. S. 1991. Dataflow Dependency Backtracking in a New CLP Language. In Proc. AAAI Spring Symposium on Constraint-Based Reasoning. Stanford. 110-127.

Havens, W. S., Sidebottom, S., Sidebottom, G., Jones, J., Cuperman, M. and Davison, R. 1990. Echidna Constraint Reasoning System: Next-generation Expert System Technology. Technical Report CSS-IS TR 90-09. The Expert Systems Laboratory, The Centre for Systems Science.

Jaffar, J. and Michaylov, S. 1987. Methodology and Implementation of a CLP System. In Proc. Fourth International Conference on Logic Programming. Melbourne, Australia.

Lloyd, J. W. 1984. Foundations of Logic Programming. Springer-Verlag, New York. 124 pages.

Mackworth, A. K. 1977. Consistency in Networks of Relations. Artificial Intelligence. 8. 99-118.

Mackworth, A. K. and Freuder, E. C. 1985. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. Artificial Intelligence. 25. 65-74.

Mackworth, A. K., Mulder, J. A. and Havens, W. S. 1985. Hierarchical arc consistency: exploiting structured domains in constraint satisfaction problems. Computational Intelligence. 1. 118-126.

Mohr, R. and Henderson, T. C. 1986. Arc and Path Consistency Revisited. Artificial Intelligence. 28. 225-233.

Nadel, B. A. 1989. Constraint Satisfaction Algorithms. Computational Intelligence. 5. 188-224.

Naish, L. 1985. Negation and Control in Prolog. In Lecture Notes in Computer Science 238, Goos, G. and Hartmanis, J. (ed.).

Older, W. and Vellino, A. 1990. Extending Prolog with Constraint Arithmetic on Real Intervals. In Proc. The Eight Biennial Conference of the Canadian Society for Computational Studies of Intelligence. Ottawa.

Ratschek, H. and Rokne, J. 1984. Computer Methods for the Range of Functions. John Wiley & Sons, Toronto.

Sidebottom, G. 1992. Projection of Numeric Constraints with Interval Domains. In Preparation.

Sterling, L. and Shapiro, E. 1986. The Art of Prolog: Advance Programming Techniques. MIT Press, Cambridge, MA.

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

Van Hentenryck, P. and Deville, Y. 1991. The Cardinality Operator: A New Logical Connective for Constraint Logic Programming. In Proc. The International Conference on Logic Programming. MIT Press. Paris, France.