National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canadä

Hierarchical Arc Consistency Applied to Numeric Processing in Constraint Logic Programming

By

Gregory Allan Sidebottom

B.Sc. (Hon.) University of Calgary

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School of Computing Science

© Gregory Allan Sidebottom 1991

Simon Fraser University

November, 1991

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author.

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-78234-X

Canadä

Approval

NAME: Gregory Allan Sidebottom

DEGREE: Master of Science (Computing Science)

TITLE OF THESIS: Hierarchical Arc Consistency Applied to Numeric Processing in Constraint Logic Programming

EXAMINING COMMITTEE:

Chair: Dr. Fred Popowich

Dr. W. S. Havens
Senior Supervisor

Dr. V. Dahl
Examiner

Dr. Alan K. Mackworth
Department of Computer Science
University of British Columbia
External Examiner

DATE APPROVED: 1991 November 13

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Hierarchical Arc Consistency Applied to Numeric Processing in Constraint Logic Programming.

Author:

(signature)

Gregory Allan Sidebottom

(name)

November 20, 1991

(date)

Abstract

There have been many proposals for adding sound implementations of numeric processing to Prolog. This thesis describes a new approach to numeric constraint processing which has been implemented in Echidna, a new constraint logic programming (CLP) language. The approach, called HACR, uses consistency algorithms which can actively process a wider variety of numeric constraints than most other CLP systems, including constraints containing non-linear functions. A unique feature of HACR 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 two advantages set HACR apart from other approaches to real number processing in CLP.

To Sue,
for making
my life complete

Acknowledgements

This work was supported by the Centre for Systems Science at Simon Fraser University, by the Alberta Research Council, and by PRECARN Associates. I would also like to acknowledge my senior supervisor, Bill Havens, who provided the initial idea for and numerous useful discussions about this work; and Miron Cuperman for implementing the algorithms described here in Echidna.

Contents

List of Figures vii

1. Introduction 1

2. Echidna Background 3

    2.1 Real Number Constraints in Echidna 4

    2.2 Overview of the Echidna Reasoning Engine 6

3. Hierarchical Arc Consistency on Reals (HACR) 8

    3.1 Domains 12

    3.2 ReviseHACR 17

    3.3 Computing Projections 23

        3.3.1 Equalities 25

        3.3.2 Inequalities 27

        3.3.3 Disjunctive Inequalities 27

4. Examples and Comparisons 28

    4.1 Polynomials and Precision 29

    4.2 Geometry 30

    4.3 Linear Equations 31

    4.4 Scheduling 31

5. Conclusions and Future Work 32

References 35

List of Figures

1. A circle described by [1] 5

2. An Echidna Program for scheduling tasks 6

3. HACR: an arc consistency algorithm for real constraints 11

4. (a) Geo-system and (b) Shore specialization hierarchies 13

5. A scheduling problem 16

6. The domain representation for a real variable S 17

7. ReviseHACR: a revision algorithm for hierarchical numeric domains 21

8. A control program 34

1. Introduction

Numeric processing has always been an important aspect of computing. But there are two major problems with traditional procedural languages using the floating point representation for real numbers. First, errors induced by floating point computations are hard to determine and analyze. Second, debugging and correctness verification for procedural languages can be very difficult. Logic programming languages, such as Prolog, address the second problem since they have well understood semantic properties (Lloyd, 1984). However, numeric processing in Prolog, as it is usually implemented with the 'is' predicate, suffers from the same problems as procedural languages using floating point numbers.

Constraint logic programming (CLP) languages (Jaffar and Lassez, 1987; Van Hentenryck, 1989) seek to add efficient algorithms for new computational domains to logic programming languages in a way that preserves their important semantic properties. However, the major CLP languages with numeric constraint processing capabilities have at least one of the three following weaknesses:

1. their applicability is limited to a small class of constraints, such as linear, polynomial or integer constraints,
2. they compute using the full precision of the underlying numeric computation implementation, whether it is needed or not, and
3. they are forced to search a large space when dealing with disjunctive constraints.

CLP languages like CLP($\mathbb{R}$) (Jaffar and Michaylov, 1987), Prolog III (Colmerauer, 1990), and CAL (Aiba et al., 1988) use symbolic constraint solving techniques. However, CLP($\mathbb{R}$) and Prolog III can only actively process 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 possible values for a variable. For consistency algorithms, a domain must be represented by some finite manipulable structure. CHIP's numeric domains are always finite integer sets. BNR Prolog's domains are real intervals and it efficiently implements many real constraints by using consistency algorithms to tighten those intervals closer to actual solutions to the constraints (Cleary, 1987; Hyvönen, 1989).

Practical consistency algorithms only partially solve constraints. Both CHIP and BNR Prolog provide search primitives which can be used to augment consistency algorithms within the logic programming (LP) language. CHIP can always find exact solutions since its consistency algorithms only deal with finite discrete sets. Search methods in BNR Prolog can tighten intervals as close to solutions as possible using the underlying finite precision computer arithmetic. This ensures that no solution for a given set of constraints is missed although sometimes answers contain no solutions.

This thesis describes an approach to real number processing in CLP, called HACR[1]. HACR 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 process a wide variety of real number constraints. The key difference is that HACR implements real domains which are disjoint sets of intervals using a hierarchical data structure. HACR 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"). HACR implements label and constraint languages which are more general than either BNR Prolog or the real number systems surveyed in (Davis, 1987). Our label language is composed of disjoint real interval sets and it contains real intervals as a sublanguage. Our constraint language contains equalities, inequalities, and disjunctions of inequalities on arbitrary expressions involving the arithmetic functions, and some expressions involving the trigonometric, exponential, root, and logarithmic functions.

HACR handles such general systems of constraints by using *partial consistency* algorithms (Nadel, 1989). Partial consistency algorithms approximate the set of solutions to a

---

[1]HACR stands for Hierarchical Arc Consistency on Real domains. The approach takes its name from the arc consistency algorithm, which is its novel component. The approach also includes case analysis algorithms which have been described by others (Mackworth, 1977; Cleary, 1987).

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. HACR uses partial consistency algorithms and case analysis algorithms to generate sets of interval tuples with one interval for each variable in a given numeric CSP. All the solutions to the CSP are contained in some interval tuple. Also, HACR is parameterized so that as it is given increasing time and space, it can usually generate intervals that converge on solution points.

The remainder of the thesis is organized as follows: Section 2 describes the aspects of Echidna which are relevant to HACR. Section 3 specifies HACR's adaptation of hierarchical arc consistency (Mackworth et al., 1985) for real number constraints. Section 4 gives some sample runs using Echidna and compares it with other CLP languages. Finally, Section 5 draws some conclusions about this research and describes some future lines of research.

2. Echidna Background

Echidna is a new type of CLP language for model-based expert systems applications. The language improves upon the limitations of existing expert system languages by combining aspects of schema-based knowledge representations, CLP, and intelligent backtracking (Havens, 1991). Echidna builds on recent advances in CLP by integrating within the language a clausal reasoning maintenance system and object-oriented knowledge structures. We believe that next-generation expert systems will incorporate richer structured knowledge representations based on object-oriented programming principles and rely on more efficient constraint propagation and dependency backtracking control structures. Echidna demonstrates that these capabilities can be combined successfully into a coherent new CLP 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). In this presentation, we augment the syntax of Edinburgh Prolog (Sterling and Shapiro, 1986) as necessary for exposition[2]. Section 2.1 describes how Echidna augments a logic programming language with real number constraints and Section 2.2 briefly describes the

[2]For clarity, we deviate from actual Echidna syntax.

SLD-resolution theorem prover (Lloyd, 1984) and arc consistency algorithms (Mackworth, 1977) used in Echidna.

2.1 Real Number Constraints in Echidna

Echidna provides domain constraints, equalities, inequalities, and disjunctions of inequalities on real number 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 $-\infty$ and $+\infty$. 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 $\{x \mid 0 \leq x \leq 1 \vee 3 < x \leq 4 \vee 7 < x < 10\}$.

Equalities and inequalities are constraints on real number 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.
```

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

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. 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]  A = 0, B = 0, R = 1
```

since three points uniquely define a circle[4]. Notice that this query results in many constraints involving non-linear expressions. HACR can restrict the domains of `A`, `B` and `R` to intervals which tightly bound this solution.

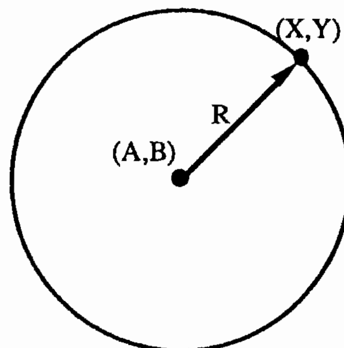

Figure 1. A circle described by [1]

HACR 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:

$$C_1 \vee C_2$$

---

[4]Actual results using Echidna for examples in this section are given in section 4.

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, HACR can deduce from the goal:

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

that `S1` is in the set $[0, 0.5] \cup [1.5, 2]$ and `S2` is in the set $[0, 0.5] \cup [2, 2.5]$[5].

2.2 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

[5]Actually, HACR deduces sets slightly larger than these sets. See section 3 for details.

associated with a domain of possible values and a set of constraints on subsets of the variables. A constraint specifies which values from the domains of its variables are compatible. A compatible set of values, one for each variable participating in a constraint, is said to *satisfy* the constraint. 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. 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[6].

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 using dependency backtracking (Havens, 1991). 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 algorithms 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 and discrete, case analysis combined with arc consistency can completely solve the CSP. If Echidna's real domains were finite and discrete, then after some finite number of iterations, `split` would have reduced all variable domains to singleton sets. However, real domains

[6]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.

are neither finite nor discrete. Currently, the number of times a variable domain is split is determined by a built-in predicate, `precision(Vars, Prec)`. Sections 3.2 and 4.1 describe its operation in more detail. We are investigating how to provide more flexible control of the case analysis methods.

3. Hierarchical Arc Consistency on Reals (HACR)

We use the notation $\nu(C)$ to denote the set of variables in the constraint C. The arity of C is $|\nu(C)|$. We assume the CSP is formulated as a directed hypergraph[7] where variables are associated with nodes and each constraint C is 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 formulated in this way, arc consistency deletes values from target variable domains which are not supported by any consistent source variable values. Such deleted values cannot be part of any global solution to the CSP. The notation Δ_X is used to denote the dynamically 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 taken as relations in the relational database model sense (Ullman, 1988). Unlike relational database theory, relations are represented intensionally and can be infinite. Variables are the attributes of an instance of a relation scheme and the relation is always restricted to values in the dynamic domains of the variables. For example, if $\Delta_X = \Delta_Y = \{1, 2, 3\}$ then the constraint C(X, Y) = 'X < Y' is the relation $\{\{(X,1),(Y,2)\}, \{(X,1),(Y,3)\}, \{(X,2),(Y,3)\}\}$, which is an instance of the 'less than' relation between X and Y. Relations are viewed as sets of mappings. Each element $\mu \in C$ can be considered a mapping from the set $\nu(C)$ of variables to the set $\bigcup_{X \in \nu(C)} \Delta_X$ of possible values for those variables. For example, for $\mu = \{(X,1),(Y,2)\} \in C$ above, $\mu(X) = 1$ and $\mu(Y) = 2$.

It will sometimes be convenient to use positional notation for constraints by giving an explicit order for the variables similar to the notation used to specify the relation scheme. If C is a constraint with $\nu(C) = \{X_1,\ldots,X_k\}$ and $a_i \in D_{X_i}$ $(1 \le i \le k)$, then the positional notation for C is $C(X_1,\ldots,X_k)$ and $C(a_1,\ldots,a_k)$ means $\{(X_i, a_i) \mid (1 \le i \le k)\} \in C$. The

[7] A directed hyper-graph is a generalization of a directed graph where hyper-arcs may 'connect' any number of nodes.

tuple $(a_1,\ldots,a_k)$ is said to *satisfy* C. If there is at least one such tuple (ie. $C \neq \emptyset$), then C is *satisfiable.*

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

[5] $$\pi_X(C) = \{a \mid (\exists\mu \in C)\ \mu(X) = a\}^8.$$

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 $X \in V(C)$, all values $a \in \Delta_X \setminus \pi_X(C)$[9] cannot be used to satisfy C since there are no corresponding values for $V(C) \setminus X$. 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. The HACR approach employs a similar but partial arc consistency algorithm, also 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

---

[8]Projection is usually defined to return a relation on some subset of the variables in the given relation. Only this special case will be needed because the algorithms described in this paper manipulate relations only by manipulating variable domains.

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

$\pi_T(C)$ 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), but it is generalized for *n*-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.

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 $T \in \nu(C') \setminus \{T'\}$ 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).

[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, so they are always inserted at the front of the queue.

```
1   procedure HACR(A):

2       procedure ReviseHACRAbstract(T, C):
3       begin
4           let Δ be such that π_T(C) ⊆ Δ ⊆ Δ_T;
5           DELETE ← (Δ ⊂ Δ_T);
6           if DELETE then Δ_T ← 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 ∈ v(C')\{T'} ∧ C ≠ C'}
15      end
16  end;
```

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

Unlike arc consistency algorithms like AC-3 and HAC, which are for finite discrete 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 3.2 describes the built-in predicate, `precision`, which is used to limit domain refinement. HACR terminates when $Q = \emptyset$, 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.

The remainder of this section is organized as follows. Section 3.1 describes how domains are represented. Section 3.2 describes how HACR's arc revision algorithm is implemented given the ability to compute $\pi_T(C)$ and section 3.3 describes how to compute $\pi_T(C)$ for most constraints.

3.1 Domains

Arc consistency algorithms usually operate on finite and discrete domains. Domains are represented extensionally as enumerated sets of possible 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. Although HAC presumes that domains are finite and discrete, 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 3.2.

Please 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 rooted directed acyclic graph (DAG). Each node is associated with a domain symbol, denoting a subset of the domain, and a mark. We distinguish between the symbol associated with each node and the domain subset which it denotes. Henceforth, the distinction is dropped. 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 DAG 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 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-DAG rooted at a particular node represents a particular subset of Δ_X. The mark for the root node of a sub-DAG indicates whether the 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-DAG rooted at the node is union of the parts represented by the sub-DAGS rooted at its children. HAC maintains the dynamic domain Δ_X by manipulating these marks[11].

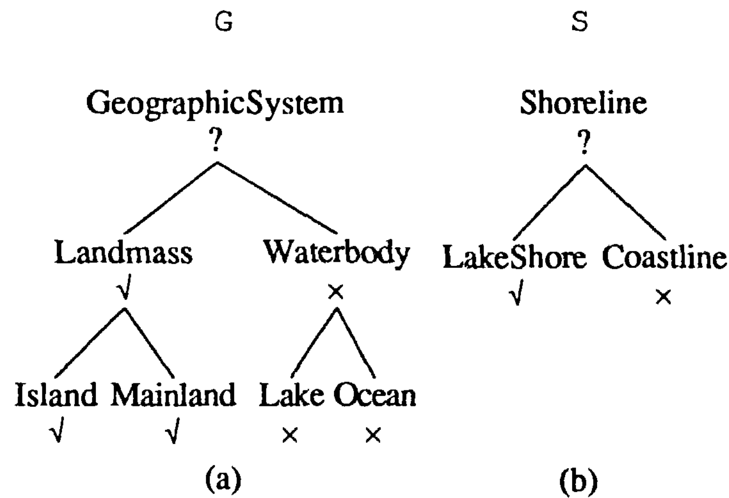

Figure 4. (a) Geo-system and (b) Shore specialization hierarchies

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 $(k+1,2s-1)$ and $(k+1,2s)$ with the conditions that:

---

[11]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.

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

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

These two conditions ensure respectively that the children cover their parent exhaustively and mutually exclusively. The leaf domains are $D_X^{mi} = \{a_i\}$ $(1 \leq i \leq 2^m)$. 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 $0 \leq k \leq m$ and $1 \leq s \leq 2^k$. The interpretation for these marks is

[8] $$M_X^{ks} = \begin{cases} \surd & \text{if } D_X^{ks} \subseteq \Delta_X \\ ? & \text{if } D_X^{ks} \not\subseteq \Delta_X \text{ and } D_X^{ks} \cap \Delta_X \neq \emptyset \\ \times & \text{if } D_X^{ks} \cap \Delta_X = \emptyset \end{cases} .$$

The dynamic domain, Δ_X, is the union of the domains of all nodes marked '√':

[9] $$\Delta_X = \bigcup \{ D_X^{ks} \mid M_X^{ks} = \surd \}.$$

However, some of the nodes marked '√' are redundant since all descendents of nodes marked '√' are also marked '√' and all descendents 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. $\Delta_X^{\surd}$ is the smallest set of domains in the tree for D_X such that $\bigcup\Delta_X^{\surd} = \Delta_X$.

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

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 a 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 and marks on nodes in subtrees rooted at nodes with domains in $\Delta_X^{\times}$ need to be change. 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 3.2 makes extensive use of these properties.

We extend hierarchical domains for real intervals as follows. The domain of each node it 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 program for scheduling (in 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.

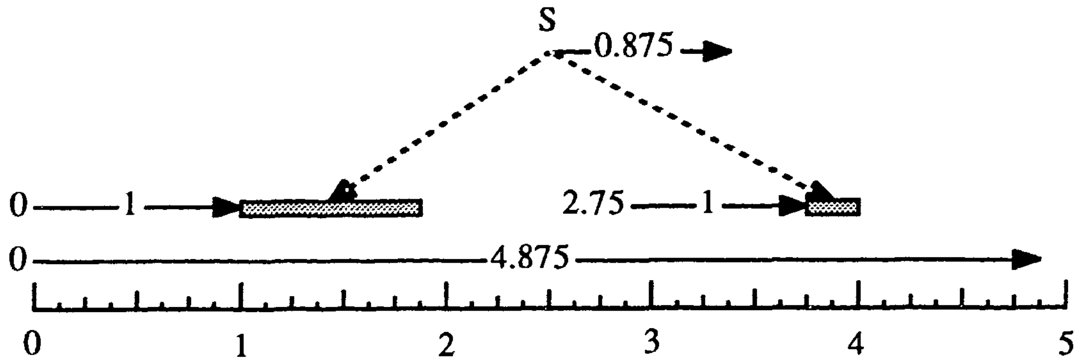

Figure 5 A scheduling problem

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 the their parent domain. The relationship between a parent and its two children is

[11]

$$\begin{array}{c} [a, b] \\ \swarrow \quad \searrow \\ [a, mid(a,b)) \quad [mid(a,b), b] \end{array}$$

where $a < mid(a,b) < b$. The types of interval bounds (open or closed) associated with a and b in the children are inherited from the parent and one of the bounds associated with $mid(a,b)$ is open while the other is closed. There are several reasonable definitions for $mid(a,b)$. If an unbounded precision (eg. rational) number system is used, then the mean $((a+b)/2)$ or the mediant (Graham et al., 1989) can be used. If a fixed precision (eg. 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) calls these two options linear and exponential splitting respectively and studies their efficiency.

Echidna presently represents interval bounds using 64-bit IEEE floating point numbers and employs linear splitting. That is, for the remainder of this thesis, we assume

[12]

$$mid(a,b) = \frac{a+b}{2}.$$

Currently, all lower bounds are closed and all upper bounds are open in order that an interval need only be stored explicitly for the root node. The interval for any other node (k, s) is calculated via its position in the tree. That is, if $D_X^{01} = [a, b)$, then

$$D_X^{ks} = \left[a + (s-1)\frac{b-a}{2^k},\ a + s\frac{b-a}{2^k} \right). \quad [13]$$

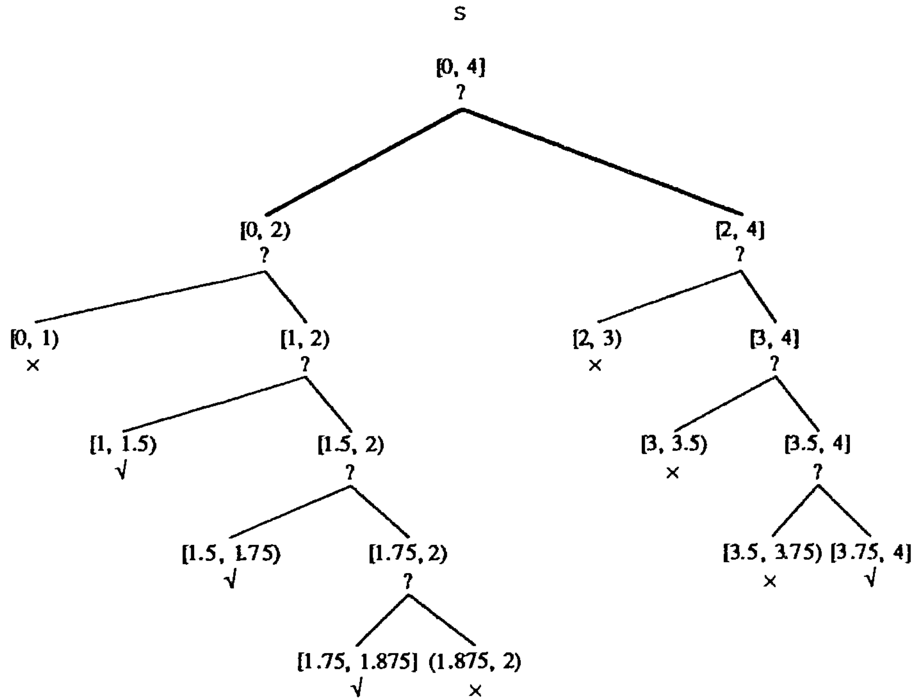

Figure 6. The domain representation for a real variable S

3.2 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:

$$[14] \qquad M_T^{ks} \leftarrow \begin{cases} \surd & \text{if } D_T^{ks} \subseteq \pi_T(C) \\ ? & \text{if } D_T^{ks} \not\subseteq \pi_T(C) \text{ and } D_T^{ks} \cap \pi_T(C) \neq \emptyset \\ \times & \text{if } D_T^{ks} \cap \pi_T(C) = \emptyset \end{cases}$$

for each subset D_T^{ks} of D_T. If all values in D_T^{ks} are consistent with some value in some $D_S^{lr} \in \Delta_S^{\surd}$, then M_T^{ks} remains '$\surd$'. Otherwise if some values are consistent with some value in some

$$D_S^{lr} \in \Delta_S^{\surd},$$

then M_T^{ks} is changed to '?' and the children domains, $D_T^{(k+1)(2s-1)}$ and $D_T^{(k+1)2s}$, are considered[12]. Otherwise none of the values are consistent so M_T^{ks} is set to '×'. By repeating this procedure for every node in $\Delta_T^{\surd}$, the new domain Δ_T is constructed according to the assignment $\Delta_T \leftarrow \pi_T(C)$. 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 $\pi_T(C)$ by generating a set of intervals whose union is $\pi_T(C)$[13]. 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 $v(C) = \{S_1, \ldots, S_{n-1}, S_n = T\}$. Iterating through $\Delta_T^{\surd}$ and searching for tuples

---

[12]Notice that for $D_T^{ks} \in \Delta_T^{\surd}$, changing M_T^{ks} from '$\surd$' to '?' implicitly removes D_T^{ks} from $\Delta_T^{\surd}$ and adds its children.

[13]Actually, when floating point numbers are used, the generated set of intervals may be a superset of $\pi_T(C)$ because the floating point approximations of the bounds of some intervals may necessarily be rounded outwardly.

$$(D^{l_1 r_1}_{S_1}, \ldots, D^{l_{n-1} r_{n-1}}_{S_{n-1}}) \in \Delta^{\surd}_{S_1} \times \ldots \times \Delta^{\surd}_{S_{n-1}}$$

which are consistent with C will be very inefficient as n 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 [14] 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, a full arc revision algorithm would not be expected to terminate. To avoid this eventuality, HACR 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.[14]

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

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

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:

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

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

---

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

[17] $$\Delta = \bigcup\{D_{\mathrm{T}}^{ks} \mid TM_{\mathrm{T}}^{ks} = \surd\}.$$

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

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 $\Delta = S \cup approx(I, \mathrm{T})$ where $approx(I, \mathrm{T})$ is the smallest superset of I which can be represented in the taxonomy for T at its current precision P_{T}. Formally, $approx(I, \mathrm{T})$ is the union of the results of intersecting I with all the intervals at nodes on level P_{T} in the taxonomy for T. That is,

[18] $$approx(I, \mathrm{T}) = \bigcup_{s=1}^{2^{P_{\mathrm{T}}}} (I \cap D_{\mathrm{T}}^{P_{\mathrm{T}}s}).$$

We generalize *approx* to apply to a finite union of intervals as follows:

[19] $$approx(\bigcup_{i=1}^{n} I_i, \mathrm{T}) = \bigcup_{i=1}^{n} approx(I_i, \mathrm{T}).$$

Section 4.3 shows how to compute a set $\{I_1,\ldots,I_n\}$ of intervals whose union is $\pi_{\mathrm{T}}(\mathrm{C})$, so we use $approx(\pi_{\mathrm{T}}(\mathrm{C}), \mathrm{T})$ to mean $approx(\{I_1,\ldots,I_n\}, \mathrm{T})$ for such an appropriate set of intervals. The approximation of $\pi_{\mathrm{T}}(\mathrm{C})$ is accumulated in Δ by repeatedly calling MarkTemp with I_i for $1 \leq i \leq n$. MarkTemp does not require that the set of intervals is disjoint.

```
1    procedure ReviseHACR(T, C):
2    begin                                                        {Δ_T = S}
3        TM_T^{01} ← ×;                                           {Δ = Ø}
4        let {I_1,...,I_n} be any set of intervals such that ∪{I_1,...,I_n} = π_T(C);
5        for i ← 1 to n do MarkTemp(D_T^{01}, I_i);               {Δ = approx(π_T(C), T)}
6        Less ← {D_T^{ks} | TM_T^{ks} < M_T^{ks}};                (* × < ? < √ *)
7        for D_T^{ks} ∈ Less do M_T^{ks} ← TM_T^{ks};             {Δ_T = S ∩ Δ}
8        return Less ≠ Ø
9    end;

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

Figure 7 ReviseHACR: a revision algorithm for hierarchical numeric domains

The ReviseHACR algorithm operates as follows. In line 3, the root temporary mark TM_T^{01} is set to '×' effectively making Δ empty. Care is taken (later in the specification of MarkTemp) to mark children of nodes with intervals in $\Delta^{\times}$ as '×' if they are ever accessed. In lines 4-5, the algorithm sets Δ to $approx(\pi_T(C), T)$ using MarkTemp, as discussed above. In line 6, 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 7, 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 6-7 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 8 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. The algorithm considers three cases (in lines 12, 13 and 14 respectively). In the first case, it returns without changing any temporary marks (in line 12) if any of the following conditions are true. If $M_T^{ks} = \times$, then $D_T^{ks} \cap \Delta_T = \emptyset$ meaning that D_T^{ks} has already been removed from Δ_T. If $TM_T^{ks} = \surd$, then $D_T^{ks} \subseteq \Delta$ meaning that an interval covering D_T^{ks} has already been generated. If $I \cap D_T^{ks} = \emptyset$ 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 13), if $D_T^{ks} \subseteq I$ (indicating that I covers D_T^{ks}) or $k = P_T$ (indicating that the precision limit of T has been reached), then D_T^{ks} is added to Δ by assigning $TM_T^{ks} = \surd$. At the precision limit, some values outside I may be added to Δ, but only ones which require greater precision to eliminate. Note that $I \cap D_T^{ks} = \emptyset$ and $D_T^{ks} \subseteq I$ can be tested efficiently by comparing appropriate bounds.

Otherwise, in the third case (starting at line 14), since the branches at lines 12 and 13 were not taken, $I \cap D_T^{ks} \neq \emptyset$ and $D_T^{ks} \not\subseteq I$. Thus, TM_T^{ks} should be set to '?' and its children should be analyzed. Lines 15-17 ensure that MarkTemp is never called recursively with

children of nodes temporarily marked '×'. If $TM^{ks}_{T} = \times$ at line 15 the children have never be accessed. This is because the root temporary mark is set to '×' in line 3 and line 16 sets the temporary mark of the current node to '?' before the recursive calls with the children. Lines 17 and 18 set the temporary marks of the children to '×' before the recursive calls on line 20, which mark the approximation of I in the subtrees rooted at the two children of D^{ks}_{T}. If k+1 is the precision limit and the point on the boundary between the two children is in I, the two recursive calls mark both children '√'. This violates the interpretation of marks. It is fixed by line 21, if necessary.

Now that we have described ReviseHACR fully, we can formally state what the HACR algorithm of figure 3 does. We say that a hyperarc (T, C) is *approximately arc consistent* if $\Delta_T = approx(\pi_T(C), T)$ and that a CSP is *approximately arc consistent* if every hyperarc in its hypergraph representation is approximately arc consistent. Upon the termination of HACR, the CSP represented by its input, A, is approximately arc consistent. It is in exactly this sense that HACR is a partial arc consistency algorithm.

3.3 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, $iso(X, X \cdot Y{=}Z)$ = '$X{=}Z{+}Y$'. 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 $f_E : \Delta_{S_1} \times \ldots \times \Delta_{S_{n-1}} \to \mathbf{R}$ to denote the function of $(S_1, \ldots, S_{n-1})$ defined by E. The range of f_E is

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

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 3.3.1 and 3.3.2 describe the computation of projections of equalities and inequalities, respectively. Section 3.3.3 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 and to reduce the complexity of HACR. 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, 1991). We consider here only computing $\pi_T(C)$ for an arbitrary constraint C subject to these two restrictions.

HACR 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:

```
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 = R²
   T2 + T4 = T5.      % (X - A)² + (Y - B)² = R²
```

where Ti are new intermediate variables ($1 \leq i \leq 5$). The domain D_{Ti} of intermediate variable Ti in a constraint Ti = 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 and:

[21] $$min\ range(f_E) = max\ range(f_E) = a.$$

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:

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

[23] $max\ \Delta_X = a_2$ 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 [22] 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 & Rokne, 1984). This analysis is applied in HACR for the arithmetic, exponential, logarithmic, root extraction, and trigonometric functions.

3.3.1 Equalities

Let C be an equality with $v(C) = \{S_1, \ldots, S_{n-1}, S_n = T\}$. By restriction 2, we can assume that n is 0, 1, 2, or 3. When $n = 0$, there are no projections to compute. For $n > 0$, *iso*(T, C) = 'T = E' and the projection $\pi_T(C)$ = *range*(f_E). If $n = 1$ then E is the constant a and *range*(f_E) = $\{a\}$. Otherwise,

[24] *range*(f_E) =

$$\bigcup \{f_E(D_{S_1}^{k_1 s_1}, \ldots, D_{S_{n-1}}^{k_{n-1} s_{n-1}}) \mid (D_{S_1}^{k_1 s_1}, \ldots, D_{S_{n-1}}^{k_{n-1} s_{n-1}}) \in \Delta_{S_1}^{\surd} \times \ldots \times \Delta_{S_{n-1}}^{\surd}\}$$

where applying f_E to the intervals $(D_{S_1}^{k_1 s_1}, \ldots, D_{S_{n-1}}^{k_{n-1} s_{n-1}})$ is defined by

[25] $f_E(D_{S_1}^{k_1 s_1},\ldots,D_{S_{n-1}}^{k_{n-1}s_{n-1}}) =$

$$\{f_E(x_1,\ldots,x_n) \mid (\exists(x_1,\ldots,x_n) \in D_{S_1}^{k_1 s_1}\times\ldots\times D_{S_{n-1}}^{k_{n-1}s_{n-1}})\}.$$

The Region Splitting theorem of Bundy (1984) ensures the correctness of this approach for computing $\pi_T(C)$. 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:

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

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

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

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

We necessarily complicate [29] by considering divisors which include zero in their domains. In this case, the quotient is the union of two disjoint intervals. For instance,

[30] $[1, 1] \div [-2, 3] = (-\infty,-1/2] \cup [1/3,+\infty).$

This is accommodated by splitting the denominator at zero:

[31] $[1,1] \div [-2,3] = [1,1] \div ([-2,0] \cup [0,3]),$

and appealing to the Region Splitting theorem which yields:

[32] $[1,1] \div ([-2,0] \cup [0,3]) = ([1,1] \div [-2,0]) \cup ([1,1] \div [0,3]).$

These two disjoint expressions can then be evaluated using [32] 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 functions of the form $f_E(D_{S_1}^{k_1 s_1}, \ldots, D_{S_{n-1}}^{k_{n-1} s_{n-1}})$ evaluated in [24] above can be reduced by combining adjacent intervals in $\Delta_{S_i}^{\surd}$ $(1 \le i < n)$. For instance, in the scheduling example of Figure 6, $\Delta_S^{\surd}$ = {[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 $\pi_T(C)$ where C is an equality with $S \in \nu(C)$. The complexity of computing $\pi_T(C)$ is another reason for decomposing equalities as described above. Even after adjacent intervals are combined, $|\Delta_{S_1}^{\surd} \times \ldots \times \Delta_{S_{n-1}}^{\surd}|$ 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 $\Delta_{S_i}^{\surd}$ is further limited by the setting of the precision for each variable (as previously described).

3.3.2 Inequalities

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

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

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

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

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 [21-23] above.

3.3.3 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 $\nu(C_1)$ and $\nu(C_2)$. If both $T \in \nu(C_1)$ and $T \in \nu(C_2)$ then values which satisfy either disjunct can be used to satisfy the whole constraint. The projection is:

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

For the other case, T only appears in one expression. Assume without loss of generality that $T \in v(C_1)$ and $T \notin v(C_2)$. 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 [21-23]. C_2 is satisfiable if and only if

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

When E_1 is the variable X and E_2 is the variable Y, $\{(X, min\ \Delta_X), (Y, max\ \Delta_Y)\} \in C_2$ so [36] is sufficient for the satisfiability of C_2. [36] is also necessary because if C_2 is satisfiable then for some $a \in \Delta_X$ and $b \in \Delta_Y$, $\{(X, a), (Y, a)\} \in C_2$. This implies $a \leq b$ and since $a \geq min\ \Delta_X$ and $b \leq max\ \Delta_Y$, [36] is true. The proofs when E_1 or E_2 are constants is simpler. Similarly, if C_2 = '$E_1 \geq E_2$' then C_2 satisfiable if and only if

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

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 $v(C_1)\backslash v(C_2) = \{X_1,\ldots,X_m\}$ 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 $T \in v(C_1)\backslash v(C_2)$, $\pi_T(C) = \Delta_T$ if C_2 is satisfiable. If C_2 is unsatisfiable, then all $\mu \in C$, when restricted to $v(C_1)$, must satisfy C_1. Thus, $\pi_T(C) = \pi_T(C_1)$.

To summarize,

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

4. Examples and Comparisons

This section provides some comparisons of Echidna's real number capabilities with other major CLP systems. The examples were run using Echidna version 1.0 on a Sun UNIX Sparcstation 1.

Each derivation of a query in Echidna induces a CSP which consists of the set of constraints selected (ie. called) at some step in the derivation. CLP languages with complete CSP solvers can answer queries exactly in terms of variable bindings for solutions to the CSP. Since Echidna's numeric constraint solving system is a partial solution to the CSP, it outputs approximate answers by binding variables to their domains

in the induced CSP after it has been made approximately arc consistent. As dicussed earlier, calls to `split` are used to further refine domains by analyzing different cases. Let C be a CSP induced by a query and let $\{X_1,\ldots,X_m\}$ be the set of variables which have appeared in a call to `split`. Then Echidna ouputs one answer for each way C can have the domain of X_i replaced by some interval on a node at level P_{X_i} in the domain taxonomy for X_i ($1 \le i \le m$) and then made approximately arc consistent[15].

4.1 Polynomials and Precision

For simplicity, only a quadratic polynomial in factored form is used in this section. The form of the polynomial affects the efficiency of the solution (Cleary, 1987). Consider the following query.

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

Echidna generates numerical solutions to polynomial equations with varying precision using the built-in predicate `precision`. We employ the call `precision([X],8)` to initially limit the precision of the variable X to 8-bits. Thus, the taxonomy for X will not be refined beyond the eight level. The call `split([X])` is used to invoke a case analysis search for solutions for X. For this query, Echidna computes the following answer (containing the two solutions, `X = 1` and `X = 2`):

```
X ∈ [0, 7.8125)        ;
no.
```

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.
```

[15] Because of the propagation floating point precision errors, it is possible for Echidna's current implementation to ouput answers which are not from partially arc consistent CSPs.

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.
```

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 query. CAL is powerful enough to find the two solutions, however.

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 solve polynomials 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 control over this facet of the computation.

4.2 Geometry

The query [2] in section 2.1, which uses the `onCircle` predicate defined by [1], can be augmented with precision and case analysis calls. The resulting query is:

```
[40] ?- 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. BNR Prolog, like Echidna, has the capability to solve this query numerically.

4.3 Linear Equations

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:

```
[41] ?- 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 exponentially with the number of variables and equations in the CSP (Cleary, 1986).

4.4 Scheduling

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

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

Echidna returns the single answer:

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

It is interesting to note that the query contains no call to the `split` case analysis predicate. In this case, HACR's consistency algorithms alone remove enough inconsistent values to split the domains into two disjoint intervals. In systems such as BNR Prolog, Prolog III and CLP(**R**), the disjunctive inequality in the `noOverlap` predicate of Figure 2 has to be expressed using nondeterminism in the program. For instance, it can be expressed as:

```
S1 ≤ S2 - D1 ; S1 ≥ S2 + D2
```

using the disjunction connective (;) of Edinburgh syntax Prolog. When this disjunctive constraint is used, the solutions to queries like [42] above are not contained in a single answer, but are distributed over several answers.

5. Conclusions and Future Work

This thesis has described how the HACR approach real number constraint processing is implemented in the Echidna CLP language. HACR supports domain constraints, equalities, inequalities, and disjunctions of inequalities on real number expressions involving arithmetic, exponential, and trigonometric functions. The set of numeric constraints supported by HACR is richer than for the other numeric constraint processing techniques cited.

HACR'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. This thesis gave a formal description of the HACR algorithm which included details about how to revise and project constraints on variables with hierarchically structured precision bounded real numeric domains. Examples were give to show 1) that HACR can be used to compute answers to varying precision under program control, 2) HACR can numerically solve some constraints which other CLP systems cannot, 3) HACR is not as efficient as other systems for simple linear constraints, and 4) HACR can process certain disjunctive constraints without resorting to case analysis algorithms.

There are at least two areas where HACR can be improved. First, consistency algorithms cannot compete with specialized symbolic constraint solving algorithms in their domain of application. Second, consistency algorithms combined with general case analysis algorithms are often insufficient to efficiently solve large complex problems.

We discuss two steps towards solving the first problem. A first step is to avoid breaking down constraints with temporary variables, wherever reasonable. In particular, with linear constraints, variables can be isolated and ranges of expressions can be computed quite easily. A second step stems from the observation that HACR's consistency algorithms never consider more than one constraint at a time. But symbolic linear constraint solving algorithms used in CLP are efficient because they combine constraints to eliminate

variables. Consequently, it would be useful to find a way to integrate some symbolic constraint solving with consistency algorithms, gaining the advantages of both approaches.

The second problem can be addressed with more programmable ways of implementing case analysis algorithms in a control meta-language. We propose a control language inspired by the `when` declarations in NU-Prolog (Thom and Zobel, 1986), which are a generalization of the `delay` declarations of MU-Prolog (Naish, 1985). We specify just enough of this proposal to write a control procedure which achieves an effect similar to the use of `split` and `precision` as they were used in section 4. The control meta language makes it possible to program case analysis algorithms directly. BNR Prolog has similar functionality, but there is no separate control program.

Instead of placing calls to control primitives such as `split` and `precision` in the logic of the program, the programmer uses `control` declarations to specify the case analysis algorithm to be used when solving goals involving numeric constraints. Control programs, like logic programs, are specified by sets of clauses, except instead of using the `:-` symbol to separate the head from the body, the symbol `control` is used. Also, control programs must not contain numeric constraints. For instance,

```
onCircle(_, c(p(A,B),R)) control
   breadthFirst([A,B,R], 16).
```

defines a control procedure for the `onCircle` predicate given in [1] in terms of the control procedure `breadthFirst`. As we will see, the control procedure

```
breadthFirst(Vars, P),
```

has the same effect as adding the calls `precision(Vars, P)` and `split(Vars)` to a query.

Logic programs and control programs are separate; they may not call each other. The control procedures associated with predicates are executed *after* a top level query succeeds but *before* the answer to the query is output. If no control procedure is specified for a goal involving variables in real constraints, a default procedure is used. A reasonable default procedure is `split(Vars)`, where `Vars` is a list of the domain variables occurring in the call to the predicate.

Control programs are executed according to the procedural interpretation of Prolog programs, by selecting clauses in the order in which they appear in the program and goals

from left to right. Control programs may also use the *or* connective (;), the *if-then* (->) connective, and arithmetic predicates (is, =:=, <, ≤, etc.) to evaluate arithmetic expressions as in Prolog.

Control programs should minimally have access to one evaluable function and two built-in control predicates which control different aspects of the search. The function call precision(X) returns the current precision of X. This function can be evaluated with the usual predicates (is, =:=, <, ≤, etc.). The control predicate setPrecision(X, P) sets the precision of X to the result of evaluating P, which is normally a function of precision(X). The control predicate case(X) removes approximately half of the values in the domain of X and introduces a choice point. Upon backtracking, it restores the deleted half and removes the other half.

The control program shown in figure 8 defines a control program using the primitives described above. BreadthFirst(Xs, P) iteratively increases precision and splits the domain of each variable in the list in a round-robin fashion until all domains are refined to precision P.

```
split(Xs, P) control
  (
   atLimit(Xs, P) -> true
  ;
   refine(Xs, P), split(Xs, P)
  ).

atLimit([], _) control true.
atLimit([X | Xs], P) control
  precision(X) =:= P,
  atLimit(Xs).

refine([], _) control true.
refine([X | Xs], P) control
  (
   precision(X) < P ->
     setPrecision(X, precision(X) + 1), case(X)
  ),
  refine(Xs, P).
```

Figure 8. A control program

This proposal could be further elaborated to include primitives which allow control programs to analyze the structure of domains, specify both the number system and the definition of *mid(x,y)*[16], and symbolically manipulate constraints. With such a powerful control language, it would be possible write a declarative specification of the problem to be solved and then fine tune problem and data specific numerical analysis and symbolic constraint solving algorithms independent of this specification.

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.

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.

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.

[16]Different number systems and definitions of *mid* were discussed in section 3.1 near formula [11].

Hyvönen, E. 1989. Constraint Reasoning Base on Interval Arithmetic. In Proc. Eleventh International Joint Conference on Artificial Intelligence, Sridharan, N. S. (ed.). Morgan Kaufmann. Detroit, MI. 1193-1198.

Jaffar, J. and Lassez, J.-L. 1987. Constraint Logic Programming. In Proc. Fourteenth ACM POPL Conf. Munich.

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.

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. 1991. 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.

Thom, J. A. and Zobel, J. 1986. NU-Prolog Reference Manual. Technical Report. University of Melbourne, Dept of Computer Science.

Ullman, J. D. 1988. Principles of Database and Knowledge-base Systems. Computer Science Press, Rockville, MD.

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