A Language
for
Optimizing Constraint Propagation

By

Gregory Allan Sidebottom

B.Sc. (Hon.) University of Calgary 1988

M.Sc. Simon Fraser University 1991

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in the School of Computing Science

Simon Fraser University

November, 1993

Approval

NAME: Gregory Allan Sidebottom

DEGREE: Doctor of Philosophy (Computing Science)

TITLE OF THESIS: A Language for Optimizing Constraint Propagation

EXAMINING COMMITTEE:

Chair: Dr. Brian Funt

Dr. William S. Havens, Senior Supervisor

Dr. Veronica Dahl, Supervisor

Dr. Fred Popowich, Supervisor

Dr. Lou Hafer, S.F.U. Examiner

Dr. Philippe Codognet, External Examiner
Institut National de Recherche en Informatique et Automatique
Rocquencourt, France

DATE APPROVED: 93/11/18

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

A Language for Optimizing Constraint Propagation.

Author:

(signature)

Greg Sidebottom

(name)

Dec 9/93

(date)

Abstract

This thesis describes projection constraints (PCs), a language for compiling and optimizing constraint propagation in the numeric and Boolean domains. An optimizing compiler based on PCs has been implemented in Nicolog, a constraint logic programming (CLP) language. In Nicolog, like other CLP languages such as CHIP, Echidna, CLP(BNR), cc(FD), and clp(FD), domains for variables are explicitly represented and constraint processing is implemented with consistency algorithms. Nicolog compiles each constraint into a set of arc revision procedures, which are expressed as PCs. Instead of using full arc revision based on enumeration, Nicolog uses regions where functions are monotonic to express arc revision procedures in terms of interval computations and branching constructs. Nicolog compiles complex constraints directly, not needing to approximate them with a restricted set of basic constraints or to introduce extra variables for subexpressions. The Nicolog compiler can handle a very general class of constraints, allowing an arbitrary mixture of integer, real, and Boolean operations with a variety of domain representations. The only requirement is that for each domain, it must be possible to compute a set of intervals whose union contains that domain. Nicolog also lets the user program using PCs directly making it possible to implement sophisticated arc revision procedures. This thesis shows that PCs are a simple, efficient, and flexible way to implement consistency algorithms for complex mixed numeric and Boolean constraints. Emperical results with a prototype Nicolog implementation show it can solve hard problems with speed comparable to the fastest CLP systems.

To my parents,
for giving me
all the advantages in life

To Sue,
for being my partner
in the achievement of great things

To Nicole,
for being my little sweetie

Acknowledgments

I would like to thank my senior supervisor, Bill Havens. It was Bill who gave me the chance to freely pursue my interests and yet gave me the direction to finish this thesis in a very short time. Bill is also to be thanked for setting up the Intelligent Systems Laboratory at Simon Fraser University. The people and equipment in the lab made doing research as much fun as mountain biking!

My supervisors, Veronica Dahl and Fred Popowich, helped in the difficult job of channelling my ideas into a coherent self contained thesis.

Jamie Andrews, Russ Ovans, and Sue Sidebottom provided comments which improved this thesis. Frederic Benhamou provided benchmark programs used for the disjunctive scheduling problem, as well as many useful suggestions. Daniel Diaz generated endless comparative data using the clp(FD) system and helped improve Nicolog's performance on several problems. Thanks also to Anto Ertl, for suggestions which helped to improve Nicolog. Finally, thanks to Martin Vorbeck for giving this thesis a most thorough reading and helping to correct so many errors.

This research was supported by the Natural Sciences and Engineering Research Council of Canada, PRECARN Associates, the Centre for Systems Science, and the Alberta Research Council.

Contents

List of Tables

List of Figures

1. Introduction

Many difficult 'real world' can be specified as finding values for a set of variables subject to a set of constraint relations. For instance, scheduling problems usually involve finding times to start a set of tasks subject to precedence and various other constraints. Packing problems consist of finding locations for objects in containers subject to the constraint that a particular space in a container can be occupied by only one object at a time, among many other constraints. These kinds of tasks are instances of what is known as the *constraint satisfaction problem* (CSP).

Problems like these have been solved with specialized algorithms implemented with procedural languages. Though these solutions can be efficient, they usually take a very long time to develop and are difficult to adapt to even small changes in the problem specification.

Van Hentenryck [89] noted that CSPs can be solved much more effectively using *constraint logic programming* (CLP) languages. CLP refers to a powerful new class of programming languages [JL87] . These languages are based on logic programming languages [Lloy84], such as Prolog. Term unification of Prolog is generalized in CLP to constraint processing in various domains. Several CLP languages have now been implemented, dealing with constraint systems involving numeric [Clea87; JM87; DVS*88; SA89; VanH89; Colm90; OV90; BO92; HSS*92; SH92; VHSD93; DC93], Boolean [BB88; SA89; Colm90; BO92; Side93; CD93], and sequence (ie. list and string) domains [Wali89; Colm90].

For solving CSPs, CLP languages have several advantages over procedural languages. Because CLP is declarative, CSPs can be expressed almost exactly as formulated. Because CLP languages have built-in backtracking, they are very nice for writing programs that generate CSP instances and solve them using search based algorithms.

CLP is suitable for solving large real world problems in a diverse set of areas. For instance, CLP has been used in digital circuit analysis [GVPZ89; Simo89], electrical engineering [HMS87], computer aided design [Jose92], financial analysis [LMY87], mechanical engineering [Jone90; NL93], and scheduling [DSVH90], just to name a few areas. Moreover, recent research [VHSD93] shows that CLP can solve difficult real world problems with efficiency comparable to specialized procedural programs.

CLP systems like CLP(R) [JM87], CAL [SA89], and Prolog III [Colm90] process constraints using symbolic manipulation algorithms, such as the simplex algorithm [DOW55], polynomial canonization algorithms [Buch85], and theorem proving algorithms [BB88; Side93]. Though these languages can solve the constraint systems for which they are designed, they have certain drawbacks. CAL is limited to polynomial constraints on real numbers, and its constraint solving

algorithm is very inefficient in the worst case. CLP(R) has efficient algorithms for linear constraints but delays solving non-linear constraints until they become linear. This means complex constraints cannot be used to prune the search space. Prolog III also delays complex numeric and list constraints, and, though it provides a complete Boolean constraint solver, it provides no mechanism to form arbitrary Boolean combinations of numeric constraints. In all these systems, the constraint solver is a black box, meaning there is no way for the user to control what it does or define new constraints not supplied with the system.

BNR Prolog [OV90] (which has evolved into CLP(BNR) [BO92]) and CHIP [DVS*88] were the first CLP systems that explicitly represented sets of possible values, also called domains, for numeric variables. Consistency algorithms refine those domains and case analysis algorithms[1] complete the search for solutions [Mack77]. BNR Prolog uses interval domains for real numeric variables and CHIP uses finite sets for integer variables. It should be noted that CHIP also uses symbolic constraint manipulation for real and Boolean variables, with drawbacks similar to those of CLP(R) and Prolog III. BNR Prolog and CHIP both open up constraint solving to user programmed case analysis algorithms. BNR Prolog, however, keeps its consistency algorithms hidden in a black box, whereas CHIP provides some user control over its consistency algorithms. In CHIP, declarations are available that allow the user to use arbitrary logic programs as active constraints[2] and, to a small degree, control how actively they are used. Unfortunately, using logic programs to write custom constraints is very inefficient when compared to built in constraints. BNR Prolog decomposes complex constraints into basic constraints using extra variables for subexpressions. Decomposing constraints is sometimes less efficient.

Newer domain based CLP systems have added improvements to the original systems. Echidna [HSS*92; SH92] takes advantage of hierarchically structured domains to control the precision of consistency and case analysis. Aristo [EK92] elaborates the CHIP user constraint system to provide more control over when these constraints are executed. CLP(BNR) allows an arbitrary mixing of Boolean and numeric constraints by treating the numbers 0 and 1 as false and true, respectively.

---

[1]By case analysis algorithms, we mean the general class of searching algorithms including backtracking and domain splitting.

[2]By 'active constraints' we mean an implementation of constraints where they are used as more than just a passive test when their arguments become ground. Active constraints should be capable of generating missing values as soon as enough information is available. For instance, the constraint $x+y=z$ should be able to instantiate a variable as soon as the other two are instantiated. An active constraint should also be able to prune the impossibilities by removing values from domains of variables.

Perhaps the greatest recent advance in domain CLP is the introduction of cc(FD) [VHSD91]. cc(FD) provides several facilities that can be used to implement active constraints efficiently. cc(FD) introduced *indexical constraints* as a way to give the user much more flexible control over consistency algorithms. The meaning of an indexical constraint depends on (ie. is indexed by) a CSP. Indexical constraints give the user a way to custom program arc revision, the procedure at the heart of arc consistency algorithms. Most of the primitive constraints in CHIP can be compiled into indexical constraints, which are like a reduced instruction set (RISC) assembly language for constraints. Indexical constraints have two advantages. First, it is possible to custom program specific constraints and constraint reasoning methods not supported by the compiler as indexical constraints, making the system considerably more flexible and extensible. Second, since indexical constraints are compiled into many simple instructions instead of a small number of complex ones, a small number of general optimizations can drastically improve the global performance of a system [DC93]. In contrast, earlier systems such as CHIP rely on a large number of specific optimizations for good performance.

cc(FD) also adds powerful ways to combine constraints, including the cardinality [VHD91], constructive disjunction, and extended ask (also known as blocking implication [VHSD93]) constraints. Cardinality constraints, which state how many constraints in a given list must be true, can be used to implement arbitrary combinations of Boolean and numeric constraints. Constructive disjunction can be used to implement very active minimum and maximum constraints, which are needed in scheduling problems. Extended ask constraints make it possible to block the addition of a constraint until truth of another constraint can be decided. A recent paper [VHSD93] shows that cc(FD) can be implemented efficiently enough to solve very difficult problems in time similar to the best special purpose programs painstakingly developed in procedural languages. clp(FD) [DC93] is an efficient implementation of a subset of cc(FD) that includes indexical constraints but omits cardinality, constructive disjunction, and blocking implication.

However, cc(FD) is more complicated than need be. Since CLP(BNR) [BO92] allows arbitrary mixing of Boolean and numeric constraints, cardinality constraints are already available. In [CD93], it is shown how to implement Boolean constraints with only indexical constraints.

In this thesis, we describe the Nicolog[3] CLP system, which is a simple way to implement a significant part of cc(FD) with comparable efficiency. Nicolog grew out of an attempt to implement the constraint system of CLP(BNR). Later, it was realized that Nicolog was using a

[3]Nicolog is pronounced ni-'kō-ˌlȯg.

generalization of indexical constraints, which we call *projection constraints* (PCs). PCs alone are sufficient to implement all the constraints available in CLP(BNR), as well as cardinality constraints and many cases of constructive disjunction and extended ask constraints. Moreover, using PCs for all classes of constraints available in cc(FD) means that, unlike cc(FD), optimizations for PCs potentially improve the performance of all constraint processing. PCs are also natural and effective for programming efficient constraint propagation methods for the complex constraints that arise in real world scheduling and configuration problems. Because of PCs, Nicolog is simpler, more flexible, and more extensible than all other CLP systems.

The remainder of this thesis is organized as follows: Chapter 2 describes how Nicolog extends logic programming with constraints and describes how these constraints are processed. Chapter 3 shows how Nicolog compiles primitive constraints through isolations to projection constraints. Chapter 4 compares Nicolog with other CLP systems. Chapter 5 gives some examples and computation results, and compares Nicolog's efficiency with some of the fastest similar CLP systems. Finally, chapter 6 closes with some conclusions and possibilities for future work.

2. The Nicolog Language

This chapter introduces the Nicolog language. Section 2.1 gives an overview of Nicolog, describing the various constraints it adds to a Prolog dialect and how they can be used. Section 2.2 describes how constraints are processed with arc consistency algorithms.

2.1 Overview of Nicolog

Nicolog contains a subset of the familiar Edinburgh family of Prologs as described in [SS86]. As is usual in CLP languages [JL87], Nicolog adds *constraints* to logic programming by introducing special predicate and function symbols. As in standard logic programming [Lloy84], uninterpreted symbols can be used for atoms and terms to be interpreted in the Herbrand domain.

Constraints fall into one of three classes: *domain*, *primitive*, and *projection* constraints (PCs). Domain constraints provide an interface between domain variables and their domains. Primitive constraints include the usual relations on Boolean and numeric expressions, as well as some less usual constraints involving arbitrary nesting of constraints and conditional expressions. As we will see in section 2.1.4 and chapter 3, all domain and primitive constraints can be expressed with PCs. In this section, we give an overview of Nicolog constraints and how they can be used. The full syntax of Nicolog is described formally in appendix A.

Currently, Nicolog only supports constraints in the integer domain. However, only domain constraints and a small part of Nicolog's implementation need to be generalized to support constraints in the real domain as well.

2.1.1 Domain Constraints

A domain constraint is of the form

term : *set*.

A domain constraint is actually not a constraint if the *set* is not ground. We will discuss this further shortly, but first let us look at the case where the set is ground. A simplest form of a domain constraint is `X:1..5`, which means the domain of `X` is the set {1,2,3,4,5}. The set can also be a union of ranges, such as `{1,3..5,7}` which means the set {1,3,4,5,7}. Expressions, such as those evaluable by the `is/2`[4] predicate, are also allowed in the definition of domains. For

[4]It is traditional in the Prolog community to refer to a predicate named `p` which takes *n* arguments as `p/`*n*.

instance, if N is instantiated to the number integer n at run time, `X:0..2^N-1` declares the domain of `X` to be $\{0,1,\ldots,2^n-1\}$.

For convenience, the first argument can be any Prolog term. In this case, all variables in the term are assigned the domain given by the set and all numbers in the term are checked for membership in the set. For instance, `[A,B,C]:1..3` sets the domains of `A`, `B`, and `C` to {1,2,3}.

In the case where the set argument is not ground, the domain constraint can be used to access the current domain of a variable. In this case, the first argument must be a single domain variable and the second argument must be either a variable or of the form `L..U` where `L` and `U` are variables. `X:D` instantiates `D` to a set term representing the domain of `X`. `X:L..U` instantiates `L` and `U` to the lower and upper bounds of the domain of `X`, respectively. These forms of the domain constraint can be used to implement case analysis algorithms including backtrack search and domain splitting [Mack77; VanH89]. For instance, domain splitting is a divide and conquer search algorithm which first tries half of a variable domain and later backtracks to try the other half. After a domain is split, consistency algorithms are applied starting with the constraints on the split variable. For finite domain variables, this process can be applied recursively until the variable is instantiated. The following predicate implements domain splitting on its argument.

```
split(X) :- X:L..L.
split(X) :- X:L..U, L < U,
    M is (U-L)/2+L,
    (X #< M ; X #>= M),
    split(X).
```

The first clause takes care of the case where the variable is instantiated, in which case its bounds are the same. In the second case, the bounds are different so the midpoint is calculated. Then, a choicepoint is set up to try `X` smaller or bigger than the midpoint. `#<` and `#>=` are inequality constraints, which are described shortly. The inequalities automatically trigger all other constraints on `X`. Finally, domain splitting is applied recursively and terminates when the variable becomes instantiated.

2.1.2 Primitive constraints

Primitive constraints include the usual numeric equalities, inequalities, disequalities (ie. $\neq$), and Boolean constraints. Numeric constraint relation symbols are prefixed with #. Thus, equality is represented by `#=`; inequalities are represented by `#<`, `#=<`, `#>`, and `#>=`; and diseqauality is represented by `#/=`. Boolean constraint relations are represented by `/\` for 'and', `\/` for 'or', `~` for 'not', `=>` for 'implies', `<=>` for 'equivalence', and `#` for 'exclusive or'. Numeric functions are represented by the standard symbols. The complete syntax for primitive constraints is given in

appendix A. Unlike many other systems, Nicolog allows non-linear constraints and constraints involving absolute value, minimum, and maximum.

Like CLP(BNR) [BO92], Nicolog represents Booleans by numbers. Thus, arbitrary nesting of constraints is possible where a nested constraint means `1` if true and `0` if false. This is a very powerful feature which allows the definition of cardinality constraints [VHD91]. For instance,

```
(X#=Y) + (A#=B) + (M#=N) #= 2
```

is true if exactly two of the subconstraints is true. Moreover, if any one becomes surely false, the other two are actively propagated. For instance:

```
?- A:1..2, B:3..4, N:4..7, (X#=3) + (A#=B) + (M#=N) #= 2.
A = _:1..2
B = _:3..4
M = _:4..7
N = _:4..7
X = 3
```

Since `A` cannot be equal to `B`, `M` is constrained to be equal to `N` and `X` is 3. If two subconstraints are surely false, then the constraint fails and backtracking is initiated.

Nicolog also allows conditional expressions in constraints. For instance,

```
cond(A, B, C) #= D
```

constrains `D` to be the same as `B` if `A` is `1` (ie. true) and constrains `D` to be the same as `C` if `A` is `0` (ie. false). In keeping with the spirit of constraint processing, this constraint can be used to propagate information in many directions. For instance, consider the following query:

```
?- C:2..3, cond(A,B,C) #= 7.
B = 7
A = 1
C = _:2..3
```

Since `C` can not be 7, Nicolog deduces that `A` must be true and `B` must be 7.

2.1.3 Tiling Rectangles with Nicolog

Before we continue with the definition of the PCs, it is a good idea to give a non-trivial Nicolog program which illustrates the use of the constraints introduced thus far. This program also illustrates the programing style used in most Nicolog programs. The program below, which was derived from the one given in [VHSD93], solves the following square packing problem (SPP):

Given:
a set of squares with given sizes and a rectangle of given size

Find:

a way to pack all the squares into the rectangle so that none overlap and there is no wasted space.

The SPP is a subproblem of a famous tiling problem [CFG91]:

Given:

a rectangle of given size

Find:

a set of squares, all of different sizes, which can be packed into the given rectangle so that none overlap and there is no wasted space.

The order of a square tiled rectangle is the number of squares packed into it. For simplicity here, we will stick to the SPP. Example problem instances, which are taken from [CFG92], are supplied by the predicate:

```
problem(N,SX,SY,Ss)
```

where `N` is a problem identification number, `SX` and `SY` are the sizes of the rectangle along the `X` and `Y` axes respectively, and `Ss` is a list of square sizes (ie. the length of each side). The following are the four problem instances we have tried with Nicolog:

```
problem(1,32,33,[18,15,14,10,9,8,7,4,1]).
problem(2,65,47,[25,24,23,22,19,17,11,6,5,3]).
problem(3,112,112,
        [50,42,37,35,33,29,27,25,24,
        19,18,17,16,15,11,9,8,7,6,4,2]).
problem(4,175,175,
        [81,64,56,55,51,43,39,38,35,33,31,30,
         29,20,18,16,14,9,8,5,4,3,2,1]).
```

It is interesting to note that problem 1 has the smallest possible order of all square tiled rectangles and problem 3 has the smallest possible order of all square tiled squares. A solution to problem 3 is shown in figure 1.

Most Nicolog programs, and in fact most CLP programs, are of the following form:

```
solveProblem(...) :-
    generateVariables(...),
    stateConstraints(...),
    stateRedundantConstraints(...),
    searchForSolution(...).
```

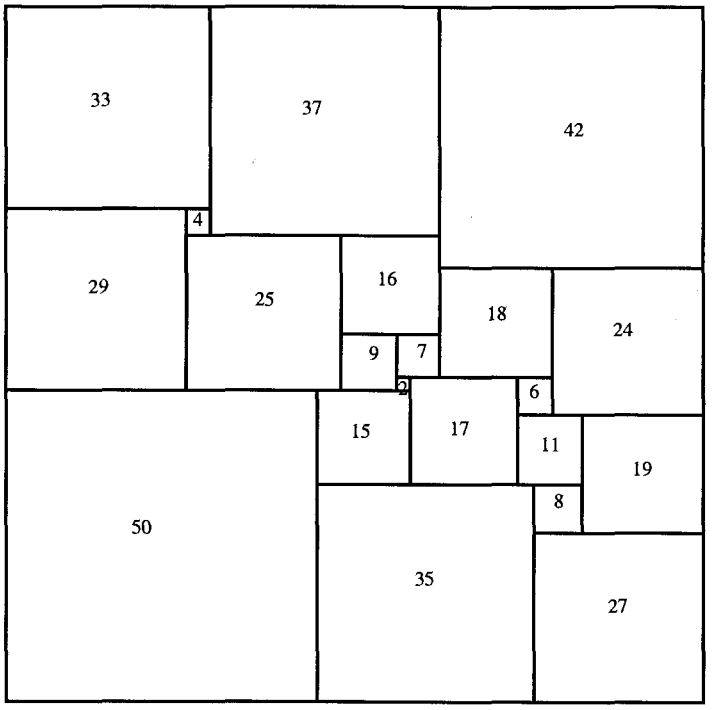


Figure 1. A square tiled square

The first goal creates the variables in the problem representation and specifies their domains. The second goal creates constraints which form a complete specification of the problem in the representation given by the variables. Since arc consistency, the constraint solving algorithm used by Nicolog, only solves the constraints approximately, the final goal is used to implement a case analysis algorithm which searches for exact solutions. The third goal creates constraints which are implied by the constraints generated by the second goal. However, redundant constraints can often drastically reduce the search space, resulting in a much faster solution time.

The top level predicate for the SPP is as follows:

```
square(P,Xs,Ys,Ss) :-
    gen(P,Xs,Ys,Ss,SX,SY),
    noOverlap(Xs,Ys,Ss),
    cap(Xs,Ss,SX,SY),
    cap(Ys,Ss,SY,SX),
    label(Xs),
    label(Ys).
```

The variable `P` is the problem number to solve. `Ss` is the list of square sizes in problem `P`, and `Xs` and `Ys` are, respectively, corresponding lists of the x and y coordinates of the lower left corners of the squares in a solution. The lower left corner of the rectangle being packed is assumed to be point (0,0). For instance, a solution to

```
?- square(3,Xs,Ys,Ss).
```

is

```
Xs = [ 0,70,33,50, 0, 0,85,29,88,93,70,65,54,50,82,54,85,63,82,29,63]
Ys = [ 0,70,75, 0,79,50, 0,50,46,27,52,35,59,35,35,50,27,52,46,75,50]
Ss = [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11, 9, 8, 7, 6, 4, 2]
```

This is the solution shown in figure 1. The first goal in `square/4` looks up the data for a given problem and generates the lists of x and y coordinates. The second goal adds constraints which require that no pair of squares may overlap. Assuming that the sum of the square areas is equal to the area of the rectangle being filled, the first two goals are sufficient to formulate the problem. However, the search can be performed much more efficiently with redundant constraints exploiting the fact that the squares must fill the rectangle exactly. The two `cap/4` goals add such capacity constraints. The final two `label/1` goals implement a case analysis algorithm which nondeterministically generates values for the goals.

We now examine the `gen/6`, `noOverlap/3`, `cap/4`, and `label/1` predicates in more detail. `Gen/6` is defined by the following two predicates[5].

```
gen(P,Xs,Ys,Ss,SX,SY) :-
    problem(P,SX,SY,Ss),
    genCoords(Xs,Ys,Ss,SX,SY).

genCoords([],[],[],_SX,_SY).
genCoords([X|Xs],[Y|Ys],[S|Ss],SX,SY) :-
    X:0..SX-S,
    Y:0..SY-S,
    genCoords(Xs,Ys,Ss,SX,SY).
```

The goal `X:0..SX-S` declares the domain of `X`, the leftmost point on the square of size `S`, to be in the integer range `0..SX-S` where `SX` is the width of the rectangle being packed. A similar goal declares the domain of `Y`, the lowest point in the square of size `S`.

`NoOverlap/3` is defined by the following predicates:

```
noOverlap([],[],[]).
noOverlap([X|Xs],[Y|Ys],[S|Ss]) :-
    noOverlap1(Xs,Ys,Ss,X,Y,S),
    noOverlap(Xs,Ys,Ss).

noOverlap1([],[],[],_X1,_Y1,_S1).
noOverlap1([X2|Xs],[Y2|Ys],[S2|Ss],X1,Y1,S1) :-
```

[5]It is customary in logic programming to use variable names starting with an underscore character for variables which occur singularly in a clause. `_SX` and `_SY` in the first clause for `genCoords/5` are examples of this.

```
    noOverlap2(X1,Y1,S1,X2,Y2,S2),
    noOverlap1(Xs,Ys,Ss,X1,Y1,S1).

noOverlap2(X1,Y1,S1,X2,Y2,S2) :-
    (X1+S1 #=< X2) \/ (X1 #>= X2+S2) \/
    (Y1+S1 #=< Y2) \/ (Y1 #>= Y2+S2).
```

For a pair of squares indentified by the coordinates (X1,Y1), (X2,Y2) and sizes S1 and S2, these predicates generate the disjunctive constraint

```
(X1+S1 #=< X2) \/ (X1 #>= X2+S2) \/
(Y1+S1 #=< Y2) \/ (Y1 #>= Y2+S2)
```

which means that at least one of the inequalities is true, guaranteeing that no pair of squares overlaps.

To improve the efficiency of the program, the cap/4 predicate adds extra "capacity" constraints. The idea is to exploit the fact that the squares must fit into the rectangle exactly. More specifically, the sum of the sizes of all squares intersecting with a vertical (horizontal) line through a given x coordinate (y coordinate) must be equal to SY (SX), the height (width) of the rectangle being packed. The following predicates add capacity constraints for each x coordinate.

```
cap(Xs,Ss,SX,SY) :-
    cap1(0,SX,SY,Xs,Ss).

cap1(SX,SX,_SY,_Xs,_Ss).
cap1(P,SX,SY,Xs,Ss) :-
    P < SX,
    sumOfSqsWith(Xs,Ss,P,SY),
    P1 is P + 1,
    cap1(P1,SX,SY,Xs,Ss).

sumOfSqsWith([],[],_P,0).
sumOfSqsWith([X|Xs],[S|Ss],P,Sum) :-
    Sum #= ((X #=< P) /\ (P #< X+S))*S + Sum1,
    sumOfSqsWith(Xs,Ss,P,Sum1).
```

A vertical line through an x coordinate P intersects with a square of size S at (X,Y) iff

$$X \leq P < X+S.$$

So cap/4 calls cap1/5 to loop for P from 0 to SX-1, the x coordinates of the vertical lines that could intersect with a packed square. For each P, cap1/5 calls sumOfSqsWith/4 to add up all the squares containing P and make sure it is SY, the height of the rectangle being packed. SumOfSqsWith adds the size of each square to the sum iff $X \leq P < X+S$. It uses the fact that Boolean false and true are represented by the numbers 0 and 1, respectively. Thus, adding up

```
((X #=< P) /\ (P #< X+S))*S
```

for each square of size S at (X,Y) computes the desired sum.

Cap/4 is called with the role of x's and y's reversed to set up the capacity constraints on y coordinates as well.

Label/1 implements the case analysis algorithm which is used to search for a placement of the squares. It uses an idea that Van Hentenryck *et. al.* [93] attribute to Aggoun and Beldiceanu [92]. At each choice point in the search, a smallest possible coordinate is identified and a square is selected to use that coordinate. Since the squares fit the rectangle exactly, there must be at least one square which fits at the smallest coordinate. To find the square for the smallest coordinate, the squares are tried in the order they are given in the problem/4 predicate. Since big squares are harder to fit than smaller ones, performance is best if they are ordered from largest to smallest. This is an example of the first fail principle [HE80].

Label/1 is implemented by the following predicate.

```
label([]).
label([X|Xs]) :-
    minlist([X|Xs],Min),
    selectSq([X|Xs],Min,Rest),
    label(Rest).
```

Minlist/2 finds Min–the smallest possible value in a non-empty list of domain variables and/or numbers.

```
minlist([X|Xs],Min) :-
    X:Min1..Max,
    minlist1(Xs,Min1,Min).

minlist1([],M,M).
minlist1([X|Xs],M1,M) :-
    X:M2..Mx,
    M3 is min(M1,M2),
    minlist1(Xs,M3,M).
```

Here, X:Min..Max is called with X bound to a domain variable or integer and with Min and Max being uninstantiated variables. If X is a domain variable, Min and Max are unified with the lower and upper bounds for the domain of X, respectively. If X is instantiated to a number, Min and Max are unified with X.

SelectSq/3 nondeterministically tries to set each of the coordinates to the minimum possible coordinate found by minlist/2. It is defined as follows:

```
selectSq([X|Xs],Min,Xs) :-
    X #= Min.
```

```
selectSq([X|Xs],Min,[X|Rest]) :-
    X #> Min,
    selectSq(Xs,Min,Rest).
```

Nicolog can solve all of the above problems in times between a few seconds for problems 1 and 2 to about 60 seconds for problem 3 and about 90 seconds for problem 4. Section 2.1.5 shows how some of the constraints can be programmed more efficiently as projection constraints. For detailed empirical results on this program, see section 5.7.

2.1.4 Projection Constraints

The basic form of a PC is

`X $=` *set*,

which means that the variable `X` is a member of *set*. There are also two forms which are abreviations for when the *set* is unbounded below or above:

`X $=<` *expr* ≡ `X $= -inf..`*expr*

`X $>=` *expr* ≡ `X $=` *expr*`..inf`

When set is ground, a PC is nothing more than a domain constraint. In fact, we could have defined a special case of the domain constraint as follows:

```
X:A..B :- integer(A), integer(B), X $= A..B.
```

However, the true power of PCs comes when domain variables occur in their *set* arguments. For instance, the expressions `<X` and `>X` denote lower and upper bounds of the domain of `X`, respectively. In general, < and > return lower and upper bounds of arbitrary *set* terms. Similarly, a variable for a *set* argument denotes an interval approximation of the domain of the variable. However, if a PC contains a variable expression then it is not executed until that variable is instantiated to an integer.

The bound access case of the domain constraint can be implemented as follows:

```
X:A..B :- var(A), var(B), A $= <X, B $= >X.
```

This illustrates the fact that it is possible to write PCs which have no logical meaning. In [VHSD91], this problem is discussed briefly. Basically, for a PC to have a logical meaning, the set denoted by the right hand side must decrease monotonically with the domains of variables in the

right hand side. This is not the case for PCs like `A $= <X` above or `X $= >A .. <B`. Chapter 3 shows how primitive constraints can be compiled into logically equivalent PCs which satisfy the monotonicity condition.

Before we describe PCs further, let us consider a simple example from [VHSD91] using the PCs we have described thus far. PCs are a way for Nicolog programmers to implement specialized constraint processing algorithms for constraints which are not handled effectively enough as primitive constraints. Suppose one needs the constraint $x \geq y + c$ where x and y are variables and c is a constant. A good way to handle this constraint is to use the following two rules:

1. whenever it is true that $y \geq k$ (for k a constant), then we would like to impose the constraint $x \geq k + c$.
2. whenever it is the case that $x \leq k$ (for k a constant), then we would like to impose the constraint that $y \leq k - c$.

This is in fact exactly what the Nicolog primitive inequality `X #>= Y + C` does. This inequality is compiled into the following two PCs.

1. `X $>= <Y + C`
2. `Y $=< >X - C`

PCs 1 and 2 correspond to rules 1 and 2, respectively.

Any time a variable is constrained to be in the empty set, the constraint means false. For instance, constraints of the form `X $= {}` and `X $=` $i..j$ with $i > j$ mean false. Constraints containing expressions which mean the fail expression `fail` also mean false.

The symbols << and >> are used in the proper implementation of strict inequalities. `<<` e means just smaller than e; it is used to implement the inequalities like `X #< 1/Y`, where some expressions do not have integer values. `<<` e is $e-1$ if e is an integer and it is $\lfloor e \rfloor$ otherwise. Similarly, `>>` e means just bigger than e, which is $e+1$ if e is an integer and $\lceil e \rceil$ otherwise.

As well as usual mathematical expressions, there are test expressions. For instance, the *and* test expression `(`e_1 `,` e_2`)` means `1` if both e_1 and e_2 are non-zero and `0` otherwise. The *equality* test expression `(`e_1 `=:=` e_2`)` means `1` if e1 and e_2 are equal and `0` otherwise.

Not only are there complements of *sets*, which are written `\` *set*, but also complements of singleton sets, written `\\` *expr*. `\` *set* means any number but the ones represented by *set* while `\\` *expr* means any number except the one represented by *expr*. It is important to note at this point that PCs

containing expressions with variables are not executed until the variables become instantiated. For instance, a PC containing `\\ X` is not executed until `X` is bound to a number.

Finally, for both expressions and sets, there are two kinds of conditionals with the same syntax and similar meanings. The first is of the form

if `->` *then* `;` *else*

which means *then* when *if* is non-zero and *else* when *if* is zero. Conditionals of the form

`b(`*bool*`,`*false*`,`*true*`,`*either*`)`

are used to implement Boolean constraints. If the set *bool* is `{0}`, then the expression means the same as *false*, if *bool* is `{1}`, then it means *true*, and if *bool* contains `{0,1}`, then it means *either*. Otherwise, the expression means `fail` in the expression case and {} in the set case. There are also two specialized Boolean conditionals for when two of the branches are the same:

`b1(`*bool*`,`*false*`,`*either*`)` ≡ `b(`*bool*`,`*false*`,`*either*`,`*either*`)`

`b2(`*bool*`,`*true*`,`*either*`)` ≡ `b(`*bool*`,`*either*`,`*true*`,`*either*`)`

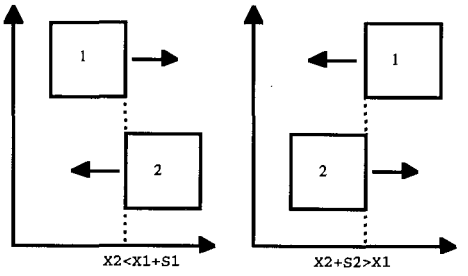
2.1.5 Square Packing Revisited

As we will see in chapter 3, PCs are sufficient to express all the primitive constraints allowed in Nicolog and CLP(BNR) [BO92]. In chapter 4, we will see that PCs can also implement cardinality and blocking implication constraints, as well as some cases of constructive disjunction constraints [VHSD93]. To see how PCs can be used to speed up programs, let us reconsider some of the constraints in the square packing program of section 2.1.3. First, consider the constraint in the `noOverlap2/6` predicate:

```
(X1+S1 #=< X2) \/ (X1 #>= X2+S2) \/
(Y1+S1 #=< Y2) \/ (Y1 #>= Y2+S2).
```

Recall that this constraint means that two squares with lower left corners at (`X1`,`Y1`) and (`X2`,`Y2`), and sizes `S1` and `S2`, respectively, do not overlap. In other words, if the two squares overlap in the *x*-axis then they must not overlap in the *y*-axis and *vice versa*. This way of handling the constraint is implemented by PCs like the following:

```
Y1$=(
      (>X2)<(<X1)+S1 , (<X2)+S2>(>X1)
      -> \ (>Y2)-S1+1 .. (<Y2)+S2-1
      ;    -inf..inf
    )
```



(a) (b)

Figure 2. Overlapping square positions on the *x*-axis

The test part of the conditional, `(>X2)<(<X1)+S1 , (<X2)+S2>(>X1)`, succeeds only if the two squares overlap in the *x*-axis. To see this, consider the first conjunct. It tests the relative positions of the squares with square 1 as far left as possible (as defined by the domains of the variables) and square 2 as far right as possible. This is depicted in figure 2a, where the arrows indicate directions the squares could be moved while still satisfying conjunct one. Similarly, the second conjunct tests the relative positions of the squares if square 1 is put as far right as possible and square 2 as far left as possible. This is depicted in figure 2b. Putting the two conjuncts together, we can see that the conjunction is true exactly when the squares overlap for any possible values for the variables.

Replacing `X`'s with `Y`'s in the test part of the conditional gives us an expression which is true if the squares overlap in the *y*-axis:

```
(>Y2)<(<Y1)+S1 , (<Y2)+S2>(>Y1)
```

By replacing bound expressions on `Y1` by `Y1` itself and rearranging, we obtain an expression which must be false for any instantiation of `Y1`:

```
Y1>(>Y2)-S1 , Y1<(<Y2)+S2
```

Negating gives an expression which must be true for any instantiation of `Y1`:

```
Y1=<(>Y2)-S1 ; Y1>=(<Y2)+S2
```

Thus, if the squares overlap on the *x*-axis, then Y1 must not be in the set :

```
(>Y2)-S1+1 .. (<Y2)+S2-1
```

In this case, the PC above constrains Y1 to be in the complement of that set. The corresponding PCs for the other variables can be constructed using similar reasoning.

Another example of the power of PCs comes from the constraint in the `sumOfSqsWith/4` predicate:

```
Sum #= ((X #=< P) /\ (P #< X+S))*S + Sum1
```

Let us consider the subconstraint:

```
B #= (X #=< P) /\ (P #< X+S)
```

which means that B is true iff P is between X and X+S. Recall that P and S are instantiated when the predicate is called, so we are only interested in PCs with B and X on the left hand side. To help understand a good way to handle the above constraint taking the instantiated variables into account, we consider the following equivalent form:

```
B #= (P-S #< X) /\ (X #=< P)
```

This way, we can see that if B is true, X is between P-S and P. Also, if B is false, X is not between P-S and P. Thus, the following PC constrains X appropriately.

```
X $= b(B,
       \ P-S+1 .. P,
         P-S+1 .. P,
       -inf..inf)
```

In some cases, we can also determine the value of B. For instance, if X can be no smaller than P-S and no larger than P, then B is 1. If X can only be smaller than P-S or it can only be larger than P, then B is 0. The following PCs enforce these facts on B.

```
B $= (
      P-S < (<X) , (>X) =< P
      -> 1
      ;  0..1
     )

B $= (
      P-S >= (>X)
      -> 0
      ;  0..1
     )

B $= (
      (<X) > P
      -> 0
      ;  0..1
     )
```

As is shown in section 5.7, replacing primitive constraints in the square packing program by the above constraints improves the execution speed of the program. Though similar constraint handling can be implemented by a combination of cardinality, constructive disjunction, and

blocking implication constraints [VHSD93], PCs provide a simple and uniform mechanism to achieve comparable execution speed.

2.2 Implementing Constraint Processing with Arc Consistency

Nicolog programs are executed by an SLD-resolution theorem prover [Lloy84] which incrementally constructs and maintains a *constraint satisfaction problem* (CSP) [Mack77]. Van Hentenryck [89] gives a complete operational semantics for CSP based CLP. He describes arc consistency [Mack77], the main constraint processing algorithm for CSPs, as an inference rule. In this section, we describe the arc consistency algorithm used by Nicolog and how it fits into a resolution theorem prover to implement CLP.

Section 2.2.1 gives some definitions and section 2.2.2 discusses arc consistency algorithms. Section 2.2.3 describes the PAC, the arc consistency algorithm used by Nicolog and section 2.2.4 gives an extended example of the operation of PAC. We conclude in section 2.2.5 with an analysis of the complexity of PAC.

2.2.1 Definitions

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 Δ_X is used to denote the domain of the variable X. 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 $a \in \Delta_X$ 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.

Nicolog manipulates the CSP primarily by using arc consistency [Mack77] to remove inconsistent values from the domains of variables under constraints. Case analysis algorithms [Mack77], such as backtracking and domain splitting, are required because arc consistency algorithms are not powerful enough to test the satisfiability of arbitrary constraints. Recall from section 2.1 that case analysis algorithms can be implemented with domain and primitive constraints. If the arc consistency algorithm ever removes all values from a variable's domain, then the constructed CSP has no solutions and the set of constraints is not satisfiable, so the theorem prover backtracks. Backtracking through a constraint consists of removing it from the CSP.

When we say C is a constraint, we mean C is a meta-variable which stands for a Nicolog primitive constraint with syntax as given in table 2 of appendix A. We will see how domain constraints and PCs fit into the picture shortly. A constraint defines a mathematical relation as follows. We use the notation v(C) to denote the set of variables in C. The arity of C is |v(C)|. We write

$C(X_1,\ldots,X_k)$ as a shorthand for $v(C) = \{X_1,\ldots,X_k\}$, specifying an order on the variables in C. If we have a constraint $C(X_1,\ldots,X_k)$ then $C(a_1,\ldots,a_k)$ is C with the numbers $(a_1,\ldots,a_k)$ substituted for $(X_1,\ldots,X_k)$. C defines the relation $\{(a_1,\ldots,a_k) \in \Delta_{X_1}\times\ldots\times\Delta_{X_k} \mid C(a_1,\ldots,a_k)\}$. During execution, Nicolog modifies the relation defined by a constraint C indirectly by changing the domains of variables in C. Where no confusion results, we treat constraints as the relations they define directly.

Later, we will take the statement '**E** is a *cterm*' to mean **E** is a meta variable standing for a constraint term with the syntax given by *cterm* in table 2 of appendix A. As with constraints, for a cterm **E**, $v(E)$ is the set of variables in **E**, $E(X_1,\ldots,X_k)$ means $v(E) = \{X_1,\ldots,X_k\}$, and $E(a_1,\ldots,a_k)$ is **E** with numbers $(a_1,\ldots,a_k)$ substituted for $(X_1,\ldots,X_k)$.

A CSP is formulated as a directed hypergraph[6] where variables are associated with nodes and each constraint C is associated with a set of arcs of the form (T, C) for each $T \in v(C)$. T is called the *target* and the rest of the variables in $v(C)$ 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. When inconsistent values are deleted from a domain, we say the domain is *refined.*

2.2.2 *Arc Consistency Algorithms*

A useful function for describing consistency algorithms is projection, denoted π, which takes as arguments a constraint $C(X_1,\ldots,X_i,\ldots,X_k)$ $(1 \le i \le k)$ and a variable X_i and returns a set of numbers. It is defined by:

$$[1] \qquad \pi_{X_i}(C) = \{a_i \mid \exists(a_1,\ldots,a_i,\ldots,a_k) \in C\}.$$

An arc (T, C) is *arc consistent* if $\Delta_T = \Delta_T \cap \pi_T(C)$. *Full* arc consistency algorithms delete all inconsistent values from every domain in the CSP, making all constraints arc consistent. *Partial* arc consistency algorithms [Nade89] delete only some inconsistent values. A well-designed partial arc consistency algorithm deletes most inconsistent values at less cost than any full consistency algorithm. Nicolog uses a partial arc consistency algorithm which is well designed for many constraints.

---

[6]A directed hypergraph is a generalization of a directed graph where hyperarcs may 'connect' any number of nodes. Most of the CSP literature deals only with binary CSPs (all constraints have arity 2 or less), which can be formulated as standard directed graphs. We use hypergraphs because we wish to deal directly with complex constraints involving an arbitrary number of variables. For simplicity, we refer to hyperarcs simply as arcs.

The fundamental operation of most arc consistency algorithms is arc *revision* [Mack77], which is implemented by a procedure Revise(T, C) where (T, C) is an arc. 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 $\Delta_T \leftarrow \Delta_T \cap \pi_T(C)$, making the arc (T, C) arc consistent. *Partial* arc revision sets Δ_T to some superset of $\Delta_T \cap \pi_T(C)$.

Full arc consistency algorithms, such as AC-3 [Mack77], call Revise repeatedly with various arcs[7]. These arc consistency algorithms terminate when there is no arc (T, C) such that Revise(T, C) can refine Δ_T further. Nicolog employs a similar but partial arc consistency algorithm, called PAC. PAC repeatedly applies a partial arc revision algorithm, called PRevise(T, C), to arcs (T, C) thereby refining Δ_T to $\Delta_T \cap approx(\pi_T(C))$, where $approx(\pi_T(C))$ is some near superset of $\pi_T(C)$ which can be computed efficiently.

PAC terminates when there is no arc (T, C) such that PRevise(T, C) can refine Δ_T further. When an arc (T, C) has the property that $\Delta_T = \Delta_T \cap approx(\pi_T(C))$, we say that it is *partially arc consistent*. Thus, PRevise(T, C) makes (T, C) partially arc consistent. To see the difference between partially arc consistent and (fully) arc consistent arcs, let us consider the constraint

```
C(X,Y,Z) = X + Y #= Z
```

with

$$\Delta_X = \{1,2\},\ \Delta_Y = \{1,4\},\ \Delta_Z = \{0,1,2,3,4,5,6,7,8,9\}.$$

The arc (Z, C) is not arc consistent because $\pi_Z(C) = \{2,3,5,6\} \neq \Delta_Z$. Revise(Z, C) deletes all six inconsistent values. Unfortunately, there is no general way to implement Revise(Z, C) that is better than summing all combinations of values for X and Y and deleting values in Δ_Z which do not appear in any sum. This can be a very expensive operation for complex constraints on many variables. Moreover, if domains are infinite, for example sets of real numbers which may be represented by sets of intervals, then enumeration of values is no longer possible.

The above discussion shows why partial arc revision is a good idea. We can delete most of the inconsistent values from Δ_Z by only performing two additions on each of the bounds of Δ_X and Δ_Y. Using the bound expressions of table 3 in appendix A for notation, we can define the computation of $approx(\pi_Z(C))$ with `<X+<Y .. >X+>Y` $= \{2,3,4,5,6\}$. With this

---

[7]Actually, the algorithms given in [Mack77] are only for constraints of arity 2, so it only applies to binary arcs. We generalize our algorithms for constraints of any arity.

implementation of *approx*(π_Z(C)), PRevise(Z, C) deletes five of the six inconsistent values at far less cost. Since many pairs of numbers could add up to 4, the one inconsistent value missed by this implementation of partial arc revision, it is usually more efficient to wait for a case analysis algorithm to make some choices to uncover this fact that 4 is not part of a solution. Only when domains are very fragmented, such as when they are the union of many small discontiguous intervals, does bound based partial arc consistency begin to suffer from its inability to exploit the 'holes.'

```
1   procedure PAC(A, N):

2       procedure PRevise(T, C): boolean
3       begin
4           compute a set approx(π_T(C)) such that π_T(C) ⊆ approx(π_T(C));
5           REFINE ← (Δ_T ∩ approx(π_T(C)) ⊂ Δ_T);
6           if REFINE then Δ_T ← Δ_T ∩ approx(π_T(C));
7           return REFINE
8       end;

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

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

2.2.3 PAC: *The Arc Consistency Algorithm Used by Nicolog*

Figure 3 presents the PAC algorithm. The input to PAC is a set A of arcs which formulate the CSP and a subset N of A. The CSP contains the constraints Nicolog has selected during an SLD-derivation. Before we look at the inner workings of PAC, let us describe the situations where it is called.

If a primitive constraint C is selected in the derivation, PAC is called with N set to the subset of the A corresponding to C. Any Herbrand variable X in C, that is a variable without an explicitly represented domain, is first submitted to the domain constraint `X:-inf..inf`, which assigns a domain as described below.

If a domain constraint of the form `X:Set` is selected, where X is a domain variable and `Set` is ground, then the assignment $\Delta_X \leftarrow \Delta_X \cap$ `Set` is performed. If Δ_X changes, PAC is called with N

containing the set of arcs with X as a source node. If X is a Herbrand variable then the assignment $\Delta_X \leftarrow$ Set is performed. In this case, PAC is not called because no constraints depend on X. If a domain constraint of the form T:Set is selected, where T is a term and Set is ground, it is handled handled as a sequence of domain constraints of the form X:Set, one for each variable X in the T. For instance, the constraint:

```
[A,B,C]:1..5
```

declares the domains of A, B, and C to be in the set 1..5.

Domain constraints of the form X:Set, where X is a variable and Set contains variables, result in the unification of Set with a term representing Δ_X. In this case, PAC is not called unless the unification involves domain variables. Unification with domain variables is described next.

The unification is generalized to include domain variables and sometimes results in calls to PAC. If two variables are unified and at least one is a Herbrand variable, then the Herbrand variable is bound to the other variable and PAC is not called. Unification of two domain variables, X and Y, is handled the same as $X:\Delta_Y$ followed by $Y:\Delta_X$. Unification of a domain variable X with a number i is handled the same as $X:i$. All other unifications with domain variables fail and result in backtracking.

PAC is also called when a PC is selected by the theorem prover. As we will see shortly, PCs correspond to a single arc where the variable T on the left hand side is the target and the rest of the variables are sources. In fact, PCs are usually a specification of how to compute $approx(\pi_T(C))$ for some constraint C. As we have seen in section 2.1.4 and will see in chapter 3, all domain and primitive constraints can be expressed in terms of PCs. If a PC is selected, then PAC is called with N set to its corresponding arc.

Now, let us look at the details of PAC. Calls to the subprocedure PRevise(T, C) on lines 2-8 refine Δ_T to

$$\Delta_T \cap approx(\pi_T(C))$$

where

$$\pi_T(C) \subseteq approx(\pi_T(C))$$

and returns true if and only if

$$\Delta_T \cap approx(\pi_T(C)) \subset \Delta_T.$$

Thus, PRevise(T, C) returns true iff and only if some inconsistent values were deleted from Δ_T. Nicolog computes the approximation $approx(\pi_T(C))$ of $\pi_T(C)$ using interval computation and branching constructs. The particular computation is dependent on the constraint C. We discuss this further in chapter 3.

Line 10 of PAC initializes Q to the set N of input arcs. The loop from line 11 to line 15 removes and revises one arc from Q in each iteration, so each arc is revised at least once. If PRevise(T, C) refines Δ_T in line 13, then Q is updated in line 14 to add just the set of arcs which could further refine their target domains. These are of the form (T', C') with $T \in v(C') \backslash \{T'\}$ and $C \neq C'$. This is because T is a source variable of C' so the 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. Arcs involving the same constraint (C = C') are not added because (T', C) is such that T' is a source variable of the arc (T, C) which was just refined. (T', C) cannot have become inconsistent because Δ_T was refined. This is because values were deleted from Δ_T precisely because there was no corresponding values for the source variables of (T, C).

By exploiting information about specific constraints, we can make PAC even more efficient by more accurately computing the set of constraints added to Q on line 14. Depending on how Δ_T was updated, some arcs may not be able to refine their target domains further even though one of their sources has changed. For instance, suppose we have the constraint

C(X,Y) = X #=< Y

with

$\Delta_X = \{0,1,2,3\}$, $\Delta_Y = \{1,2,3\}$.

If some other constraint causes the deletion of 0 and 1 from Δ_X, then PRevise(Y, C) can delete 1 from Δ_Y. However, if 3 is deleted from Δ_X then (Y, C) is still arc consistent and should not be added to Q. Furthermore, if 2 is also deleted from Δ_X then (Y, C) is arc consistent even if the domains are further refined. Therefore, (X, C) and (Y, C) should not be added to Q until backtracking restores some values. To avoid adding arcs to Q in these situations, they can be augmented with *trigger* and *satisfiability* tests [EK92]. Trigger tests determine if an arc stays consistent even if a value has been deleted from one of its source domains. A reasonable trigger test in the example above would block (Y, C) from being added to Q unless the lower bound of Δ_X changes. Satisfiability tests determine if an arc is consistent for all values left in its domains. A satisfiability test in the above example could block (Y, C) from being added to Q when the upper bound of Δ_X is smaller than the lower bound of Δ_Y. The addition of these tests are a simple way to

make the AC-3 based PAC algorithm approximate the more efficient and complex AC-4 [MH86] and AC-5 [DVH91] arc consistency algorithms.

Nicolog automatically derives reasonable trigger tests from the form of a PC as follows. If the PC refers to <X or >X, then it is triggered on changes to the lower or upper bound of Δ_X, respectively. If a variable X occurs in a set expression (*set* in table 3 of appendix A), then it is triggered on changes to either the lower or upper bound of Δ_X. If a variable X occurs in an expression (*expr* in table 3) then the PC is not triggered until X is instantiated to a constant. For simplicity in the following discussion, we assume PAC is implemented as specified in figure 1, without trigger tests.

To show PAC is guaranteed to terminate, we observe that all computer implementations of domains must be finite and that PRevise never increases the size of a domain. Thus, since arcs are added to Q only when PRevise reduces the size of a domain, PRevise can only refine a domain a finite number of times[8]. PAC terminates when Q = Ø, the exit condition on line 11. Otherwise, the loop of lines 11-15 is executed. Line 12 deletes one arc from Q. New arcs 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 PRevise, at some point no arcs will be added to Q. Thus, Q eventually becomes empty and PAC terminates.

2.2.4 *An Example Run of* PAC

In this section, we give an example run of the PAC. This should give the reader a better understanding of how PAC operates. The following also gives a gentle introduction to the content of chapter 3 by giving a concrete example of how approximate projections can be:

1. computed with interval reasoning, and
2. represented with PCs.

To get a feel for how PAC operates, consider the following CSP:

```
A:0..10, B:1..9, C:0..9,
B #> C,
A #= 9*B + C
```

[8]For infinite domains, computer representations usually place restrictions on how many times they can be refined. For instance, floating point intervals cannot be refined when their two endpoints are adjacent floating point numbers, except to collapse them to a singleton set or the empty set. [SH92] gives another approach which associates a precision parameter with domains to limit the number of times they can be refined.

Here, we use domain constraints such as `A:1..10` as a short way of declaring the domain of a variable, ie. $\Delta_A = \{1,2,\ldots,10\}$. As we will see in chapter 3, each primitive $C(X_1,\ldots,X_k)$ can be compiled into k PCs, one for each variable. Basically, this is done by first isolating each variable and then computing a set expression corresponding to the expression on the other side. For instance, the isolations of `A #= 9*B + C` are

```
A #= 9*B+C,
B #= (A-C)/9, and
C #= A-9*B.
```

These are compiled into:

```
N = {
     A $= 9*(<B)+(<C)     .. 9*(>B)+(>C),
     B $= ((<A)-(>C))/9   .. ((>A)-(<C))/9,
     C $= (<A)-9*(>B)     .. (>A)-9*(<B)
    }.
```

In chapter 3, we will see that the conjunction of the PCs in **N** is equivalent to `A #= 9*B + C`. Also, we will see that the set denoted by the right hand side of these expressions is an interval approximation of projecting the constraint onto the variable in the left hand side. Similarly, `B #> C` is equivalent to

```
A = {
     B $> (<C),
     C $< (>B)
    }.
```

For this example, we will use the 5 PCs above to represent arcs with the targets being the variables on the left hand sides. Consider the call **PAC**(A∪N, N)

Line 10 initializes **Q** to **N**, so we start with

```
Q = {
     A $= 9*(<B)+(<C)     .. 9*(>B)+(>C),
     B $= ((<A)-(>C))/9   .. ((>A)-(<C))/9,
     C $= (<A)-9*(>B)     .. (>A)-9*(<B)
    }.
```

For concreteness, suppose arcs are deleted on the top at line 12 and added on the bottom at line 14. So the first call of **PRevise** is with `A $= 9*(<B)+(<C)..9*(>B)+(>C)`. Evaluating the set expression, we calculate:

```
9*(<B)+(<C)  ..  9*(>B)+(>C) =
9*1+0  ..  9*9+9 =
9..90.
```

Thus, we can use 9..90 for the value of *approx*(π_A(A#=9*B+C)). Intersecting `9..90` with Δ_A in **PRevise** changes Δ_A to `{9,10}` so **PRevise** returns true. Since the only arcs which have `A` as a source are already present, no arcs are added to **Q** at line 14 and we end up in the following state:

```
A:9..10, B:1..9, C:0..9

Q = {
      B $= ((<A)-(>C))/9  ..  ((>A)-(<C))/9,
      C $= (<A)-9*(>B)    ..  (>A)-9*(<B)
    }.
```

To revise `B $= ((<A)-(>C))/9..((>A)-(<C))/9` we calculate

```
((<A)-(>C))/9  ..  ((>A)-(<C))/9 =
(9-9)/9  ..  (10-0)/9 =
0..10/9 =
{0, 1}. (since this is an integer range)
```

Intersecting `{0, 1}` with Δ_B in **PRevise** changes Δ_B to `1` so **PRevise** again returns true. This time, two arcs not already present have `B` as a source: `A $= 9*(<B)+(<C)..9*(>B)+(>C)` and `C $< (>B)`. However, only the second is added to **Q** at line 14 since the first was compiled from `A#=9*B+C`, the same constraint as the one just revised. Thus, we get the following state for the next iteration:

```
A:9..10, B:1, C:0..9

Q = {
      C $= (<A)-9*(>B)    ..  (>A)-9*(<B),
      C $< (>B)
    }.
```

To revise `C $= (<A)-9*(>B)..(>A)-9*(<B)` we again calculate a range using bounds:

```
(<A)-9*(>B)  ..  (>A)-9*(<B) =
9-9*1  ..  10-9*1 =
{0, 1}.
```

Intersecting `{0, 1}` with Δ_C in **PRevise** changes Δ_C to `{0, 1}` so **PRevise** returns true again. As before two arcs have C as a source, but only one was compiled from a different constraint, so the state for the next iteration is:

```
A:9..10, B:1, C:0..1
```

```
Q = {C $< (>B), B $> (<C)}.
```

In revising `C $< (>B)`, it is easy to see that Δ_C should be updated to `0`. This causes two equality arcs to be added to Q, resulting in the following state:

```
A:9..10, B:1, C:0

Q = {
      B $> (<C),
      A $= 9*(<B)+(<C)    .. 9*(>B)+(>C),
      B $= ((<A)-(>C))/9  .. ((>A)-(<C))/9
    }.
```

Revising `B $> (<C)` succeeds without any changes to Δ_B. In the next iteration, revising `A $= 9*(<B)+(<C)..9*(>B)+(>C)` changes Δ_A to `9`. Since all the arcs with A as a source involve the same constraint, no new arcs are added to Q and the next state is:

```
A:9, B:1, C:0

Q = {B $= ((<A)-(>C))/9 .. ((>A)-(<C))/9}.
```

The final arc revision succeeds with no changes, so PAC terminates and, in this case, manages to find the single solution to the CSP.

2.2.5 *Complexity of* PAC

The computational complexity of binary arc consistency algorithms has been widely studied [MF85]. However, there is little information about the complexity of arc consistency algorithms for constraints of arity greater than two. For a CSP, let n be the number of variables, let e be the number of constraints, let k be the maximum constraint arity, let d_T be the number of constraints on variable `T`, and let a be the maximum number of possible domain refinements. For finite domains, a is usually the cardinality of the largest domain.

For binary CSPs, the complexity of an enumeration-based Revise algorithm is $\mathbf{O}(a^2)$ and the complexity of AC-3 is $\mathbf{O}(a^3e)$ [MF85]. Unfortunately, the situation is not so good for these algorithms when they are generalized to the k-ary constraint case. Since Revise enumerates the Cartesian product of k domains, it is $\mathbf{O}(a^k)$. The generalization of AC-3 for k-ary constraints is shown in lines 9-16 of figure 3, except Q is initialized to A, since it is not an incremental algorithm like PAC. However, the worst case occurs when many arcs are added to Q each time line 14 is executed. The total number of arcs added to Q on line 14 in the course of execution can be far greater than the number of arcs in A. Thus, we may ignore the fact that Q is initialized to N instead of A in our worst case analysis.

The following analysis of the number of arc revisions for k-ary AC-3 like algorithms generalizes that given in [MF85]. The worst case occurs when Revise makes the smallest domain refinement possible each time it is called and, moreover, when none of the arcs to be subsequently added to Q is already in it. Arcs are added to Q when a call Revise(T, C) succeeds in refining Δ_T. In this case, at most $(d_T - 1)$ arcs are added to Q. That number may be entered a times per variable, so the total number of arcs added to Q is:

$$\sum_{T \in vars(\text{CSP})} a(d_T - 1) = a(ke - n)$$

Regardless of whether Revise refines a domain, exactly one arc is deleted on each iteration so the number of iterations is at most the original size of Q plus the number of arcs added to Q during the run, which is in $\mathbf{O}(a(ke - n)) = \mathbf{O}(ake)$. Since each iteration calls Revise, the complexity of AC-3 generalized for k-ary constraints is $\mathbf{O}(a^{k+1}ke)$.

AC-3 is exponential in the number of possible domain refinements because Revise blindly enumerates domains. With PRevise, we can exploit interval computations to do much better. As we will see in chapter 3, PRevise only needs to evaluate an expression on intervals which contain variable domains, which takes $\mathbf{O}(k)$. Thus, PAC is $\mathbf{O}(ak^2e)$ for many classes of constraints. These complexity results are similar to those found in [DVH91], except here we generalize to numeric constraints of arity greater than two and use a consistency algorithm derived from AC-3 instead of the more efficient AC-4.

3. Compiling Primitive Constraints into Projection Constraints

Van Hentenryck, Saraswat, and Deville [91] indicate that primitive constraints can be implemented with PCs, but they do not describe how to do this in general. In this chapter, we show how all of the primitive constraints available in Nicolog can be compiled into PCs. Furthermore, our compilation method does not require that complex constraints be decomposed into simple basic constraints by introducing extra variables. For instance, in many systems, a constraint such as `A*B+C#=D` would be decomposed into the conjunction `A*B#=T, T+C#=D`, introducing a new variable `T`. In some cases, it is more efficient to reason with the original constraint directly.

As suggested by previous discussions, Nicolog compiles a primitive constraint by first symbolically processing it to produce a set of equivalent constraints with a variable isolated on one side of the constraint predicate symbol and a cterm[9] on the other. Then, isolations are used to compile PCs which implement the approximate projection procedures used in the domain update operation of PRevise. Evaluating cterms on domains of variables instead of specific values in variable domains roughly corresponds to projecting constraints. Evaluating cterms on intervals containing variable domains makes it possible to efficiently compute good approximate projections using interval computation methods [AH83; Bund84] combined with branching constructs.

Section 3.1 formally defines isolations and explains the relationship between them and projections of constraint relations. Section 3.2 describes how to compute isolations for various classes of constraints. Section 3.3 shows how to translate isolations into PCs which implement domain update procedures. On the right hand side of these PCs are set valued expressions which denote the approximate projections of constraints. Approximate projections are computed by evaluating a cterm on intervals containing variable domains. Sections 3.4 and 3.5 describe how numeric and Boolean cterms, respectively, are compiled to PC set expressions which implement their evaluation on interval domains.

3.1 Isolations and the Relationship with Projections

Nicolog compiles a constraint C by first producing a set of *isolations*, usually one isolation of the form `X` *r* E for each variable `X` $\in$ v(C). In `X` *r* E, *r* is a constraint symbol (eg. `#=`, `#<`, etc.) and `X` *r* E is equivalent to C. For instance, the example in section 2.2.4 represented arcs as isolations of the target variables. Thus, isolations of `A#=9*B+C` are {`A#=9*B+C`, `B#=(A-C)/9`, `C#=A-9*B`}, and isolations of `B#>C` are {`B#>C`, `C#<B`}.

---

[9]Recall that a *cterm* is a constraint term in the syntax given in table 2 of appendix A.

The isolations of a primitive constraint C have a relationship with its projections which tells us how to compile C into a set of equivalent PCs. The exact relationship depends on whether the constraint symbol used in C is an equality, disequality, or inequality. For all constraint symbols, the relationship with projections can be defined in terms of the evaluation of a cterm $E(X_1,\ldots,X_k)$ with sets of numbers instead of particular numbers substituted for the variables. For sets of numbers $S_1,\ldots,S_k$, let

[2] $$E^*(S_1,\ldots,S_k) = \{E(a_1,\ldots,a_k) \mid (a_1,\ldots,a_k) \in S_1\times\ldots\times S_k\}.$$

To describe the relationship between isolations and projections, let

$$C(X_1,\ldots,X_i,\ldots,X_k)$$

be a constraint which has isolations of the form

[3] $$X_i\ r_i\ E(X_1,\ldots,X_{i-1},X_{i+1},\ldots,X_k)\ (1 \le i \le k)$$

where r_i is one of the constraint symbols (#=, #/=, #=<, #<, #>=, or #>)[10], and let

$$S = E^*(\Delta_{X_1},\ldots,\Delta_{X_{i-1}},\Delta_{X_{i+1}},\ldots,\Delta_{X_k}).$$

Thus, S is the set of all possible values for E given the current domains of its variables. If the isolation is an equality (ie. r_i is #=) then:

[4] $$\pi_{X_i}(C) = S.$$

Informally, we can see this is true because the constraint is equivalent to X_i #= E. The projection onto X_i is precisely values for X_i that can make this equality true, which is the same as S, the set of possible values for E. For a formal proof, we need to take care to note the difference between a constraint and the relation it defines. Recall from section 2.1.1 that a constraint $C(X_1,\ldots,X_i,\ldots,X_k)$ together with the domains of its variables defines a relation $\{(a_1,\ldots,a_k) \in \Delta_{X_1}\times\ldots\times\Delta_{X_k} \mid C(a_1,\ldots,a_k)\}$. Since section 2.1.1, we have been abusing notation by letting C stand for $\{(a_1,\ldots,a_k) \in \Delta_{X_1}\times\ldots\times\Delta_{X_k} \mid C(a_1,\ldots,a_k)\}$. Having raised this point, we will continue to not distinguish between constraints and the relations they define. For the following proof, the main consequence of this simplification is that the statement $(a_1,\ldots,a_k) \in C$ implies $a_j \in \Delta_{X_j}$ $(1 \le j \le k)$.

---

[10]Boolean constraints have one of the Boolean constraint symbols (/\, \/, ~, =>, <=>, or #) as their principal functor. Before compilation, each Boolean constraint B is replaced by B #= 1 so we only have to deal with isolations using #=, #/=, #=<, #<, #>=, or #> during compilation.

The formal proof of [4] follows from the fact that the following four statements are equivalent:

[5] $$a_i \in \pi_{X_i}(C),$$

[6] $$(\exists a_1 \in \Delta_{X_1}, \ldots, \exists a_{i-1} \in \Delta_{X_{i-1}}, \exists a_{i+1} \in \Delta_{X_{i+1}}, \ldots, \exists a_k \in \Delta_{X_k})\ (a_1,\ldots,a_i,\ldots,a_k) \in C,$$

[7] $$(\exists a_1 \in \Delta_{X_1}, \ldots, \exists a_{i-1} \in \Delta_{X_{i-1}}, \exists a_{i+1} \in \Delta_{X_{i+1}}, \ldots, \exists a_k \in \Delta_{X_k})\ a_i = E(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_k),$$

and

[8] $$a_i \in S.$$

Using [1] (the definition of projection in section 2.2.2), we can show that [5] is equivalent to [6]. To show [5] implies [6], suppose [5] is true. Then by [1] there is a tuple $(a_1,\ldots,a_i,\ldots,a_k)$ such that $(a_1,\ldots,a_i,\ldots,a_k) \in C$. Thus, since $a_j \in \Delta_{X_j}$ $(1 \leq j \leq k)$, the existence of the tuple $(a_1,\ldots,a_i,\ldots,a_k)$ such that $(a_1,\ldots,a_i,\ldots,a_k) \in C$ implies [6]. Conversely, suppose [6] is true. Then since $(a_1,\ldots,a_i,\ldots,a_k) \in C$, [1] shows us that [6] implies [5].

To show [6] is equivalent to [7], we first note that in this case [3] is an equality of the form X_i #= E and it is equivalent to C. To show [6] implies [7], suppose [6] is true. Then we have a tuple $(a_1,\ldots,a_i,\ldots,a_k)$ such that $(a_1,\ldots,a_i,\ldots,a_k) \in C$. Since C is equivalent to X_i #= E, $(a_1,\ldots,a_i,\ldots,a_k) \in C$ implies [7]. Conversely, suppose [7] is true. Then, again by the equivalence of the constraint with the isolation, we know that $(a_1,\ldots,a_i,\ldots,a_k) \in C$. Since, $(a_1,\ldots,a_i,\ldots,a_k) \in C$ implies the values come from the appropriate domains, $(a_1,\ldots,a_i,\ldots,a_k) \in C$ implies [6].

To show [7] and [8] are equivalent, first suppose [7] is true. Then by using [2], we can see that [7] implies $E(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_k) \in S$. This implies [8] is true. Conversely, suppose [8] is true. Then since $S = E^*(\Delta_{X_1},\ldots,\Delta_{X_{i-1}},\Delta_{X_{i+1}},\ldots,\Delta_{X_k})$, there exists values in the appropriate variable domains such that $a_i = E(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_k)$. Thus, [7] is true.

The projection is slightly more complicated if the isolation [3] is an inequality, for instance, if r_i is #=<. It depends on whether S is closed or open above as follows:

[9] $$\pi_{X_i}(C) = \begin{cases} \{x \mid x \leq sup\ S\} \text{ if } S \text{ is closed above} \\ \{x \mid x < sup\ S\} \text{ if } S \text{ is open above} \end{cases}$$

Here, *sup S* is the least upper bound of *S*. That is, the number x such that for all $y \in S$, $y \leq x$ or ∞ if *S* is not bounded above. Note that *sup S* = *max S* for sets which are closed above but *max S* is undefined if *S* is open above[11]. We can establish [9] by showing that [5] and [6] are equivalent to

[10] $$a_i \leq \mathsf{E}(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_k)$$

and that [10] is equivalent to

[11] $$a_i \in \{x \mid x \leq sup\ S\}$$

if *S* is closed above, or [10] is equivalent to

[12] $$a_i \in \{x \mid x < sup\ S\}$$

if *S* is open above. [5] and [6] are shown equivalent above. The proofs for the remaining equivalences are similar to those for equalities, except the proofs are split into different cases for when *S* is closed and open above. Thus, we will not give them in as much detail.

Since [3], which is X_i #=< $\mathsf{E}(\mathsf{X}_1,\ldots,\mathsf{X}_{i-1},\mathsf{X}_{i+1},\ldots,\mathsf{X}_k)$ is equivalent to C, we can see that [6] and [10] are equivalent. By [2], $\mathsf{E}(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_k) \in S$. Thus, if *S* is closed above, then $a_i \leq$ *sup S*. So in this case, [10] and [11] are equivalent. Otherwise, *S* is open above and $a_i <$ *sup S*. So [10] and [12] are equivalent in this case as well.

When r_i is #<, the projection is simply:

[13] $$\pi_{\mathsf{X}_i}(\mathsf{C}) = \{x \mid x < sup\ S\}.$$

The proof is the same as when r_i is #=<, except [10] is a strict inequality and it only needs to be proven equivalent to [12].

#>= and #> use the lower bound *inf S*, which is number x such that for all $y \in S$, $y \geq x$ or $-\infty$ if *S* is not bounded below. They are as follows:

The proofs are analogous to the #=< and #< cases, respectively.

[14] $$\pi_{\mathsf{X}_i}(\mathsf{C}) = \begin{cases} \{x \mid x \geq inf\ S\} \text{ if } S \text{ is closed below} \\ \{x \mid x > inf\ S\} \text{ if } S \text{ is open below} \end{cases}$$

---

[11]It is important to note here that a set can be bounded yet still open above. For instance $[0,1) = \{x \mid 0 \leq x < 1\}$ is bounded above by one and also open above since it does not contain one. This complication is only relevant for real domains, since integer domains are always closed above if they are bounded above.

[15] $$\pi_{X_i}(\mathsf{C}) = \{x \mid x > \inf S\}.$$

If r_i is the disequality symbol $\#/=$, then the projection is:

[16] $$\pi_{X_i}(\mathsf{C}) = \{-\infty\ldots\infty\}\backslash\begin{cases} S & \text{if } |S| = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

where $\{-\infty\ldots\infty\}$ is the extended set of extended real numbers, including the two infinities. In the case that S is not a singleton, [16] gives $\pi_{X_i}(\mathsf{C}) = \{-\infty\ldots\infty\}$. This is correct because there is at least two tuples for the variables $(X_1,\ldots,X_{i-1},X_{i+1},\ldots,X_k)$, say $(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_k)$ and $(b_1,\ldots,b_{i-1},b_{i+1},\ldots,b_k)$. Thus, every value a_i is in $\pi_{X_i}(\mathsf{C})$ since if $a_i = \mathsf{E}(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_k)$ then we have $a_i \neq \mathsf{E}(b_1,\ldots,b_{i-1},b_{i+1},\ldots,b_k)$ and vice versa.

In the case that S is a singleton, we can prove that [16] is correct by showing that statements [5], [6],

[17] $(\exists a_1 \in \Delta_{X_1}, \ldots, \exists a_{i-1} \in \Delta_{X_{i-1}}, \exists a_{i+1} \in \Delta_{X_{i+1}}, \ldots, \exists a_k \in \Delta_{X_k})\ a_i \neq \mathsf{E}(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_k),$

and

[18] $$a_i \in \{-\infty\ldots\infty\}\backslash S$$

are all equivalent.

[5] and [6] are shown equivalent above. Since [3] is equivalent to C, we can see that [6] and [17] are equivalent. If $|S| = 1$ then $S = \{\mathsf{E}(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_k)\}$. Thus $a_i \notin S$ and [17] is equivalent to [18].

It should be noted that since projection equations [4,9,13-16] apply to isolations of the form $X_i\ r_i$ $\mathsf{E}(X_1,\ldots,X_{i-1}, X_{i+1},\ldots,X_k)$, it is the case that $X_i \notin v(\mathsf{E})$. We call isolations with this property *independent*. As discussed in the next section, it is not always possible to find independent isolations. Fortunately, *dependent* isolations (where $X_i \in v(\mathsf{E})$) have a useful relationship with projections as well. It is possible to show that the projection equations are true for dependent isolations if we change the equalities (=) to subset or equal ($\subseteq$). Thus, with dependent isolations, the projection equations define how to approximate projections and still give useful information about how to implement PAC. In fact, using dependent isolations is a step towards decomposing complex constraints by introducing intermediate variables for subexpressions.

3.2 Computing Isolations

It is well understood how to isolate variables in constraints with common mathematical functions. The Mathematica [Wolf91] function `Isolate[]` can be used to find independent isolations for a wide variety of constraints. The Nicolog compiler implements some of the functionality of the Mathematica `Isolate[]` function so it can find reasonable but not necessarily independent isolations for all equalities and disequalities, and most inequalities. Nicolog puts constraints in a canonical form and then finds isolations by applying inverse functions to both sides of a constraint (changing the direction of inequalities as appropriate) until one side is an isolated variable.

The canonization and isolation process is straightforward for constraints involving only addition, subtraction, multiplication, division by a number, and raising to a positive integer power. These constraints are put in the form `(`*x* `+ (`*y* `* (X^`*n* `* ...)) + ...)` *r* `0`, where *r* is a constraint symbol, *x* and *y* are numbers, *n* is a positive integer, and `X` is a variable with a positive domain. Finding the isolations from a constraint of this form is a simple matter of subtracting, dividing, and extracting roots on both sides until the variables are reached. Here and in all cases below, the most efficient approximate projection procedures will be constructed if the isolation cterms are simplified to minimize their complexity. This usually means minimizing the number of function applications.

If all the variables occur singly in the canonical form, then the isolations are independent. Constraints with linear terms are a special case of this. The problem of polynomial constraints, which may have more than one independent isolation per variable, is handled by Nicolog with dependent isolations which result in approximate projections.

Division by more general cterms (instead of just numbers) is allowed. To handle this, Nicolog first puts the constraint in the form E/F *r* G/H. Then the above procedure is applied on E*H *r* G*F. In this case, care must be taken to avoid problems which arise when the denominator could be zero.

Other functions, such as `min`, `max`, `abs`, the Boolean valued numeric comparison functions (like nested equalities, etc.) and Boolean functions have very different algebraic properties. Many of their inverses are not even functions, but relations. Nicolog implements a general way of finding inverses with these varied functions. First, the canonical form is generalized to allow variables or cterms involving these functions where only variables were allowed previously. This extended canonical form requires that all the arguments to the newly included functions (ie. `min`, `max`, etc.) are also in canonical form. Secondly, a general method of specifying inverses for all functions is introduced (see below). With this inverse specification method, it is possible to produce an

isolation for every variable occurrence in every constraint that does not have problems such as the possibility of division by zero and the possibility of inequality direction change.

For any function f which takes n arguments, we denote the inverse of f on the ith argument by f_i. In other words,

$$f(X_1,\ldots,X_i,\ldots,X_n) \ \#= \ Y$$

is equivalent to

$$f_i(Y,X_1,\ldots,X_{i-1},X_{i+1},\ldots,X_n) \ \#= \ X_i.$$

Note that $+_1 = +_2 = -$, $-_1 = +$, $-_2 = -$, $*_1 = *_2 = /$, etc. Note also that many of these new inverses are not functions. For instance, $\min_1(0,0)$ can be any non-negative number since min(X,0) #= 0 is true with X replaced by any nonnegative number.

For a cterm $E(X_1,\ldots,X_k)$ possibly containing these new inverses, we generalize the definition of $E^*(S_1,\ldots,S_k)$ recursively as follows. If E is a number x, then $E^* = \{x\}$. If $E(X_1,\ldots,X_k)$ is a function application $f(E_1,\ldots,E_n)$, let Y_i be a variable with $\Delta_{Y_i} = E_i^*(S_1,\ldots,S_k)$ $(1 \le i \le n)$ and Z be a variable with $\Delta_Z = [-\infty,+\infty]$. Then $E^*(S_1,\ldots,S_k) = \pi_Z(Z \ \#= \ f(Y_1,\ldots,Y_n))$.

3.3 From Isolations to Projection Constraints

We can now start to define the function *proj* which takes an isolation

$$\mathsf{Iso} = (X_i \ r \ E(X_1,\ldots,X_{i-1},X_{i+1},\ldots,X_k)).$$

Proj Iso returns a PC denoting a procedure which implements the assignment

$$\Delta_{X_i} \leftarrow \Delta_{X_i} \cap approx(\pi_{X_i}(C)).$$

If Iso is equivalent to the argument constraint C in an arc (X_i, C) passed to the PRevise procedure of figure 3, *proj* Iso implements the assignment required in line 6 of PRevise.

If r = '#=' then

$$proj \ \mathsf{Iso} = (X_i \ \$= pr \ E)$$

where *pr* is a function which translates cterms (syntax in table 2 of appendix A) to sets (syntax in table 3). We will define *pr* for the cterms allowed in Nicolog shortly. The result of evaluating *pr*

E is the closed interval $[a, b]$[12] which is an approximation of $\mathsf{E}^*(\Delta_{X_1},\ldots,\Delta_{X_{i-1}},\Delta_{X_{i+1}},\ldots,\Delta_{X_k})$. Using equation [4] , we see that $[a, b] = approx(\pi_{X_i}(\mathsf{C}))$. Operationally, X_i \$= *pr* E means

$$\Delta_{X_i} \leftarrow \Delta_{X_i} \cap [a, b].$$

So executing X_i \$= *pr* E performs $\Delta_{X_i} \leftarrow \Delta_{X_i} \cap approx(\pi_{X_i}(\mathsf{C}))$ as expected.

If *r* = '#=<' then

proj lso = (X_i \$=< *pr* E).

Assuming again that the result of evaluating *pr* E is $[a, b]$, executing X_i \$=< *pr* E is the same as performing

$$\Delta_{X_i} \leftarrow \Delta_{X_i} \cap [-\infty, b].$$

Using equation [9], we see that $[-\infty, b] = approx(\pi_{X_i}(\mathsf{C}))$, so executing *proj* lso again performs as expected. Similarly, if *r* = #< then

proj lso = (X_i \$< *pr* E).

X_i \$< *pr* E means

$$\Delta_{X_i} \leftarrow \Delta_{X_i} \cap [-\infty, <<b].$$

This definition is justified by equation [13]. The definitions of *proj* lso for *r* = #>= and *r* = #> are

proj lso = (X_i \$>= *pr* E) and

proj lso = (X_i \$> *pr* E),

respectively. If *pr* E = $[a, b]$, then they mean

$$\Delta_{X_i} \leftarrow \Delta_{X_i} \cap [a, +\infty] \text{ and}$$

$$\Delta_{X_i} \leftarrow \Delta_{X_i} \cap [>>a, +\infty],$$

---

[12]For simplicity in this presentation, we only use closed bounds. This is fine for integer domain systems such as the present implementation of Nicolog. For true real domain systems, open bounds can also be useful. It is possible to generalize this work to include open bounds. This gives the system more power for dealing with strict real inequalities at the expense of a more complicated implementation. *Outward rounding* [Clea87; BO92] is applied when the computations of *a* and *b* are not exact. Our << and >> functions round outward in the integer domain and could be generalized to round outward so they could be used with real domains as well.

respectively. These two definitions are justified by equations [14] and [15], respectively.

If $r = \#/=$ then

```
proj lso = (Xi $=
            ( (<pr E) =:= (>pr E) ->   \ pr E
              ;                        -inf..inf)).
```

Since `(<pr E) =:= (>pr E)` is true iff E denotes a singleton set, using [16], we can see that

```
            (<pr E) =:= (>pr E) ->   \ pr E
            ;                        -inf..inf
```

means $\pi_{X_i}(C)$ exactly. So once again, executing *proj* lso again performs

$$\Delta_{X_i} \leftarrow \Delta_{X_i} \cap approx(\pi_{X_i}(C)),$$

as expected.

3.4 Interval Computation, Monotonic Regions, and Numeric Functions

This section defines the function *pr* which translates a cterm $E(X_1,\ldots,X_n)$ to a set term denoting an approximation of $E^*(\Delta_{X_1},\ldots,\Delta_{X_n})$ when E contains the numeric functions and Boolean valued numeric comparison functions allowed in Nicolog constraints.

All cterms may contain numbers and variables. For a number *n*, *pr n* = *n*. With variables, Nicolog approximates each domain Δ_X with [*inf* Δ_X,*sup* Δ_X]. Thus, Nicolog only uses domain bounds to compute approximate projections and ignores any 'holes' in domains which could be created by disequalities or certain ways of handling disjunctive constraints or division by values that could be zero, *et cetera* [SH92][13]. Thus, for a variable X, *pr* X = <X. .>X.

3.4.1 Arithmetic Functions

For expressions containing numeric functions, Nicolog uses interval computation with branching constructs. Bundy [1984] gives a general theory of functions applied to intervals whereas Alefeld

---

[13]We note here that it is not difficult to use the results of this paper to implement consistency algorithms like HACR [SH92], which do account for holes in domains. It is a matter of applying projections to the sets of intervals containing the domains of source variables and accumulating the union of the results for intersection with the target domain.

and Herzberger [1983] give some specific results for the arithmetic functions. The following formulas from Alefeld and Herzberger define the four arithmetic operations on intervals:

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

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

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

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

Formulas [19] and [20] yield very efficient interval computation procedures. Bundy's theory provides a way to improve the efficiency of [21] and [22], a way to deal with [22] when zero is in the denominator, and a theory to handle any function, all by analyzing the monotonicity properties of functions. Given a function f of n arguments to an n-tuple $(I_1,\ldots,I_n)$ of intervals, Bundy shows how to compute the set

$$f^*(I_1,\ldots,I_n) = \{f(x_1,\ldots,x_n) \mid x_1 \in I_1,\ldots,x_n \in I_n\}.$$

He calls a tuple of intervals a *region*. A function is *simply monotonic* in a region if it is monotonicly increasing or decreasing for each of its arguments for all values in that region. More formally, simply monotonic is defined as follows. A function f is simply monotonic on a region $(I_1,\ldots,I_n)$ if for all i $(1 \le i \le n)$, for each $x_j \in I_j$ $(j \ne i)$ and each $x,y \in I_i$ such that $x < y$ either f is *monotonicly increasing* in argument i:

$$f(x_1,\ldots,x,\ldots,x_n) < f(x_1,\ldots,y,\ldots,x_n)$$

or f is *monotonicly decreasing* in argument i:

$$f(x_1,\ldots,x,\ldots,x_n) > f(x_1,\ldots,y,\ldots,x_n).$$

Bundy's theory gives a way to apply f^* to a region when f is simply monotonic in that region. Addition and subtraction are simply monotonic in all regions. Multiplication and division are simply monotonic in regions which do not contain 0. The result of applying f^* to a region R where f is simply monotonic is an interval which can be computed from the bounds of R. The theory further states that f^* can be applied to a region where f is not simply monotonic by splitting the region into smaller regions where f is simply monotonic, applying f^* to each of those regions, and taking the union as the result.

Both Cleary [87] and Benahmou and Older [92] use the term *interval convex* which is related to the term *simply monotonic* as follows. A constraint $C(X_1,\ldots,X_k)$ is interval convex if Δ_{X_j} is an

interval ($1 \le j \le k$) and for all i ($1 \le i \le k$), given the independent isolation X_i *r* E(X_1,...,X_{i-1},X_{i+1},...,X_k) of C, E is simply monotonic in the region (ΔX_1,...,ΔX_{i-1},ΔX_{i+1},...,ΔX_k). Our definition generalizes previous ones, which only apply to basic constraints involving at most one function symbol, such as `A + B #= C` and `A * B #= C`.

Since addition and subtraction are simply monotonic in all regions, Nicolog compiles them into projection terms using the following definitions:

pr (A+B) = (<*pr* A) + (<*pr* B) . . (>*pr* A) + (>*pr* B)

pr (A-B) = (<*pr* A) - (>*pr* B) . . (>*pr* A) - (<*pr* B)

These definitions implement [19] and [20] exactly. Bundy [84] explains these definitions in terms of monotonicity properties. The intuition behind computing ranges for monotonic functions is this: If a function is monotonicly increasing in an argument A, then the lower (upper) bound of the range should be obtained using <*pr* A (>*pr* A). If a function is monotonicly decreasing in an argument A, then the lower (upper) bound of the range should be obtained using >*pr* A (<*pr* A). Addition and subtraction are monotonic for all arguments, addition increases in both arguments, and subtraction increases in its first and decreases in its second argument.

Equations [21] and [22] suggest the following general translations for multiplication and division:

```
pr (A*B) =
    Bs = [ (<pr A) * (<pr B) , (<pr A) * (>pr B) ,
           (>pr A) * (<pr B) , (>pr A) * (>pr B) ] ,
    min Bs..max Bs

pr (A/B) =
   (<pr B) =<0  ,  (>pr B) >=0 ->     -inf..inf
   ;                            Bs = [ (<pr A) / (<pr B) , (<pr A) / (>pr B) ,
                                       (>pr A) / (<pr B) , (>pr A) / (>pr B) ] ,
                                     min Bs..max Bs
```

For convenience, these two definitions name common subexpressions using =. The translation for division tests if zero is in the denominator and returns the set of all numbers if so. In Nicolog, `-inf` and `inf` are represented simply by the minimum and maximum numbers allowed in the system. Thus, constraints involving possible division by zero usually do nothing until the domain refinements eliminate that possibility.

Though these definitions are correct, it is possible to construct much more efficient translations by using Bundy's [84] theory to analyze domains of expressions using knowledge about constant values, functions, and compile time declared domains. For instance, since multiplication is

increasing in positive arguments, if it is known that A and B are positive, multiplication can be simplified to:

$$pr\ (A*B) = \quad (<pr\ A) * (<pr\ B)\ ..\ (>pr\ A) * (>pr\ B)$$

Division increases with positive first arguments and decreases with positive second arguments, so if A and B are positive, division can be compiled to:

$$pr\ (A/B) = \quad (<pr\ A) / (>pr\ B)\ ..\ (>pr\ A) / (<pr\ B)$$

The general definition for division above only tests if the smallest interval containing denominotor domain contains zero. However, it is possible that zero has been deleted from the domain, in which case the projection is the union of two disjoint intervals. The PCs currently implemented in Nicolog do not support the detection of holes in domains nor the propagation of those holes. However, PCs could be generalized for use with the the hierarchical arc consistency algorithm decribed in [SH92] to deal more effectively with domains which are sets of disjoint intervals.

The actual definitions Nicolog uses for multiplication and division are multiple conditional expressions which test for various simply monotonic regions and resort to the general definitions only in the general cases. These definitions can be found in appendix B. Partially evaluating tests with compile time information about ranges of expressions can determine that some tests are always true or false. In that case, a conditional construct can be replaced by the appropriate branch. A special case is for linear equations which, when isolated, only result in multiplication or division by constants. For instance, consider the constraint A #= 9*B + C from the example in section 2.2.4. Its isolations are

```
A #= 9*B+C,
B #= (A-C)/9, and
C #= A-9*B.
```

These isolations are compiled into

```
A $= 9*(<B)+(<C)  ..  9*(>B)+(>C),
B $= ((<A)-(>C))/9  ..  ((>A)-(<C))/9, and
C $= (<A)-9*(>B)  ..  (>A)-9*(<B),
```

respectively. Implementing the constraint with these PCs causes PAC to behave as shown in section 2.2.4.

As a consequence of these definitions and partial evaluation, when constraints only apply functions in their simply monotonic regions, Nicolog compiles domain update functions which are efficient in the sense that they only need to evaluate the isolation cterms twice. In this case, PRevise takes

$\mathbf{O}(k)$ and PAC takes $\mathbf{O}(ak^2e)$ time as discussed in section 2.2.4. Moreover, PAC computes the same result as AC-3 if domains are all intervals, meaning no values have been deleted between their bounds. In this case, projections are computed exactly and PAC is a full (ie. non-partial) arc consistency algorithm like AC-3, except PAC can also handle non-finite integer and real interval domains as well. There are other consistency algorithms with these properties [VanH89;DVH91], but for less general classes of constraints.

Most other common numeric functions, such as exponentiation, root extraction, and the trigonometric functions, can be handled with techniques similar to those used here for division [SH92]. These techniques are most effective when the functions are monotonic in many classes of arguments.

3.4.2 Conditional Expressions and Comparisons

Next, we consider the conditional expression function `cond`. There are various possible ways to translate applications of this function. For Nicolog, we chose a way that gives enough power to do complex reasoning like that shown in the example at the beginning of section 2.1.2. Of course, if a different implementation of the `cond` primitive is needed, it can always be programmed with PCs. Here is the definition used in Nicolog[14]:

```
pr cond1(D,B,C) =
      DneqB = ((<pr D)>(>pr B) ; (>pr D)<(<pr B)),
      DneqC = ((<pr D)>(>pr C) ; (>pr D)<(<pr C)),
      ( DneqB ->     (DneqC -> {} ; 0)
       ; DneqC ->    1
       ;             0..1)

pr cond2(D,A,C) =
      b2(pr A, pr D, -inf..inf)

pr cond3(D,A,B) =
      b1(pr A, pr D, -inf..inf)

pr cond(A,B,C) =
      b(   pr A,
           pr C,
           pr B,
           min[(<pr B),(<pr C)]..max[(>pr B),(>pr C)])
```

[14]Recall from the end of section 2.1.4 that b1(*bool,false,either*) ≡ b(*bool,false,either,either*) and b2(*bool,true,either*) ≡ b(*bool,either,true,either*) are specialized Boolean conditionals for when two of the branches are the same.

The translation of $cond_1$ is quite complicated. The subexpression DneqB is true if D and B range over disjoint sets and thus cannot be equal. A similar statement holds for DneqC. The fail set, {}, which causes backtracking, is the result if D can be equal to neither B nor C. If D can be equal to only B, then the result is true and if D can be equal to only C then the result is false. The translations for $cond_2$, $cond_3$ and cond are straightforward, considering that they test if A has to be true or false.

The #= functions are compiled as follows:

```
pr (C #=1 B) =
      b(   pr C,
           (<pr B) =:= (>pr B) -> \ pr B ; -inf..inf,
           pr B,
           -inf..inf)

pr (C #=2 A) = pr (C #=1 A)

pr (A #= B) =
        (<pr A) =:= (>pr A), (<pr B) =:= (>pr B) ->
                                        (<pr A) =:= (<pr B)
      ; ((<pr A)>(>pr B) ; (>pr A)<(<pr B)) ->      0
      ;                                             0..1
```

The translation for $\#=_1$ tests if C has to be true or false, and acts like an equality projection atom if true, a disequality projection atom if false, and does nothing otherwise. Since #= is symmetrical, $\#=_2$ is the same as $\#=_1$. The translation for #= returns true if the arguments have to be the same, false if the arguments can not be the same, and both Boolean values (0..1) otherwise. These definitions allow equality reasoning such as the following:

```
?-  A:1..5, B:6..10, C:1..10,
    (A#=B)#=(C#=5).
A = _:1..5,
B = _:6..10,
C = _:{1..4,6..10}
```

Since Δ_A and Δ_B are disjoint, A and B cannot be equal and A#=B is false, Thus, C#=5 is false as well. Consequently, 5 is deleted from Δ_C.

The #/= functions are compiled using similar reasoning as follows:

```
pr (C #/=1 B) =
      b(   pr C,
           pr B,
           (<pr B) =:= (>pr B) -> \ pr B ; -inf..inf,
           -inf..inf)
```

```
pr (C #/=2 A) = pr (C #/=1 A)

pr (A #/= B) =
        ((<pr A) >= (>pr B) ; (>pr A) =< (<pr B)) ->          1
      ; (<pr A) =:= (>pr A), (<pr B) =:= (>pr B) ->
                                        (<pr A) =\= (<pr B)
      ;                                                 0..1
```

The translations for the #=< functions are as follows:

```
pr (C #=<1 B) =
     b(   pr C,
          >>(<pr B)..inf,
          -inf..(>pr B),
          -inf..inf)

pr (C #=<2 A) =
      b(  pr C,
          -inf..<<(>pr A),
          (<pr A)..inf,
          -inf..inf)

pr (A #=< B) =
        (>pr A) =< (<pr B) ->        1
      ; (<pr A) > (>pr B) ->         0
      ;                              0..1
```

The translations of #=<1 and #=<2 test if C is true or false, and act like the appropriate inequality projection atoms. The translation for #=< returns true if the arguments have to be less than or equal to, false if the arguments cannot be less than or equal to, and both Boolean values otherwise. The translations for the other inequality functions are similar.

Given these definitions, Nicolog can do quite sophisticated reasoning for fairly complex constraints. For instance, consider the following query:

```
?- E1:1..10, E3:1..4, E4:5..7,
   (E1 #=< 5)*2 + (E3 #=< E4)*2 #=< 3.
E4 = _:{5..7}
E3 = _:{1..4}
E1 = _:{6..10}
```

Nicolog reasons that E1 #=< 5 must be false, since E3 #=< E4 is true for any values in the domains of E3 and E4. If E1 #=< 5 were true, then the two products would sum to 4, but the sum has to be less than 4. So since E1 must be greater than 5, values below 5 are removed from its domain.

3.4.3 Absolute Value, Minimum, and Maximum

Now that we have defined translations for the `cond` and Boolean valued comparison functions, it is easy to define the translations for the `abs`, `min`, and `max` functions in terms of these functions. For instance, `cond(A #>= 0, A, -A) #= B` is equivalent to `abs(A) #= B`. Unfortunately, Nicolog does not find independent isolations for the `cond` formulation, so its direct translation results in 3 projection atoms for each occurrence of the variable `A`. Moreover, each projection atom is fairly complicated and weak in its domain refinement capability. For instance, we have:

```
?- B:3..10, cond((A #< 0), -A, A) #= B.
A = _:-inf..inf
B = _:3..10
```

If AC-3 were applied to this constraint, the answer would be

```
A = _:{-10..-3,3..10}
```

Fortunately, we can compile more efficient PCs which also get this result. Nicolog uses the following translations for the `abs` functions:

pr (abs$_1$(B)) =
    {*pr* -B, *pr* B}

pr (abs(A)) =
    (<*pr* A) >= 0 -> *pr* A
    ; (>*pr* A) < 0 -> *pr* -A
    ; 0..max[-(<*pr* A), (>*pr* A)]

The translation of abs$_1$(B) is simply the union of the ranges of -B and B. This gives absolute value constraints more power:

```
?- B:3..10, abs(A) #= B.
A = _:{-10..-3,3..10}
B = _:3..10
```

Note that the definition of *pr* (abs(A)) makes it possible to revise domains even when A ranges over both positive and negative numbers. For instance:

```
?- A:-3..2, abs(A) #= B.
A = _:-3..2
B = _:0..3
```

`Min` can also be implemented with `cond`, giving the following behavior:

```
?- A:5..10, B:4..11, cond((A #< B), A, B) #= C.
A = _:5..10
B = _:4..11
C = _:4..11
```

However, we can see that C cannot really be any bigger than 10, the largest value for A. To do this kind of reasoning, we use the following definitions:

```
pr (min1(C,B)) =
        (>pr min1(C,B)) =< (<pr B) -> pr C
      ;                               -inf..inf

pr (min2(C,A)) =
        (>pr min2(C,A)) =< (<pr A) -> pr C
      ;                               -inf..inf

pr (min(A,B)) =
        (>pr A) =< (<pr B) ->   pr A
      ; (<pr A) >= (>pr B) ->   pr B
      ;                         min[(<pr A),(<pr B)]..
                                min[(>pr A),(>pr B)]
```

The apparently circular definition for *pr* (min1(C,B)) in terms of itself is actually an abuse of notation. Recall that min1(C,B) #= A means the same as min(A,B) #= C, so A is the first argument to the minimum function. The definition for min1 above depends on whether A is known to be less than B. Since A is not present in this context, we use min1(C,B) to refer to A. Similarly, we use min2(C,A) to refer to B in the definition of *pr* (min2(C,A)). To actually implement this, Nicolog keeps track of cterms for both sides of a primitive constraint. While recursively descending into one side, Nicolog applies inverse functions to the other side so it can be referred to when necessary. For instance, to compile a PC for the variable X using the constraint

```
min(X+Y,Z) #= W
```

the cterm involving X is isolated in two steps:

```
X+Y #= min1(W,Z)

X #= min1(W,Z)-Y.
```

In the first step, we can see that min1(W,Z) is equal to X+Y. When the second constraint isolating X is compiled, we need to evaluate *pr*(min1(W,Z)). To do this, we substitute X+Y for min1(W,Z) in the definition of min1 above.

The translations for the `min` function check if the relationship between the arguments has to be =< or >= and do the appropriate thing in each case. Using these definitions, we get:

```
?- A:5..10, B:4..11, min(A,B) #= C.
A = _:5..10
B = _:4..11
C = _:4..10
```

The translations for the `max` functions are similar.

3.5 Boolean Functions

Boolean functions have been implemented by using `min` for `/\`, `max` for `\/`, and `1-B` for `~B` [BO92]. Codognet and Diaz [93] implement Boolean constraints with PCs using arithmetic only. For instance, they use

pr(C /\1 B) = (<C) .. (>C) * (>B) +1- (<B).

However, Nicolog uses a specialized approach for Boolean constraints which more directly corresponds to the Boolean propagation rules.

Boolean cterms are compiled by Nicolog using a four branch generalization of binary decision diagrams (DDs) [Brya86; Brya92]. A DD is a labeled rooted directed acyclic graph representing a Boolean cterm B. We use *label*(D) to denote the label of the root of a decision diagram D. DDs are either *terminal* or *nonterminal*. A terminal DD consists of a single terminal root node. A nonterminal DD consists of a root node connected to four descendant DDs. Terminal DD roots and nonterminal DD branches are labeled by one of {Ø,{0},{1},{0,1}}, each of the possible Boolean domains. Nonterminal DD roots are labeled by cterms. Associated with each nonterminal DD D there is a function *branch*$_D$ which takes a Boolean domain *B* and returns the descendant DD connected to the branch labeled by *B*. As with cterms, we take $D(X_1,\ldots,X_n)$ to mean D is a DD containing variables $X_1,\ldots,X_n$.

For each Boolean cterm $B(X_1,\ldots,X_n)$ we can construct a DD $D(X_1,\ldots,X_n)$ such that

$$B^*(\Delta_{X_1},\ldots,\Delta_{X_n}) = D^*(\Delta_{X_1},\ldots,\Delta_{X_n})$$

where $D^*(\Delta_{X_1},\ldots,\Delta_{X_n})$ is defined by the following two rules:

1. If D is terminal then $D^*(\Delta_{X_1},\ldots,\Delta_{X_n}) = label(D)$.
2. If D is nonterminal with *label*(D) = E, let

$$B = \mathsf{E}^*(\Delta_{X_1},\ldots,\Delta_{X_n})) \cap \{0,1\}.$$

Then

$$\mathsf{D}^*(\Delta_{X_1},\ldots,\Delta_{X_n}) = branch_{\mathsf{D}}(B)^*(\Delta_{X_1},\ldots,\Delta_{X_n}).$$

Informally, this means each nonterminal tests a cterm $\mathsf{E}(X_1,\ldots,X_n)$ (usually a variable, but not always with mixed constraints) by evaluating $\mathsf{E}^*(\Delta_{X_1},\ldots,\Delta_{X_n})$. The branch labeled with $\mathsf{E}^*(\Delta_{X_1},\ldots,\Delta_{X_n}) \cap \{0,1\}$ leads toward the terminal labeled with result of evaluating B^* on the domains of its variables. Following the path from the root of a DD to a terminal according to the tests gives the result of evaluating B^*. For instance, suppose

```
B(A,B,C,D) = A \/ B/\C \/ D
```

and we wish to compile the PC which projects `E #= B` onto `E`. This PC is of the form

```
E $= pr B
```

where evaluating *pr* B is equal to $\mathsf{B}^*(\Delta_A, \Delta_B, \Delta_C, \Delta_D)$. If we depict a DD with circles around nonterminal labels, rectangles around terminal labels, and labeled lines directed from top to bottom for branches, an appropriate DD is shown graphically in figure 4. To calculate $\mathsf{B}^*(\Delta_A, \Delta_B, \Delta_C, \Delta_D)$ we follow the path from the root node at the top to a terminal node at the bottom. At each nonterminal, we take the branch labeled by the domain of the variable labeling the node. Eventually, we reach a terminal which is labeled with the result. Since empty domains cause backtracking immediately, we omit the branches on Ø, which all lead to a terminal labeled with Ø. An explicit Ø terminal is never needed for the usual Boolean functions. However, an explicit Ø terminal is needed for the inverses of some Boolean cterms, as we will see.

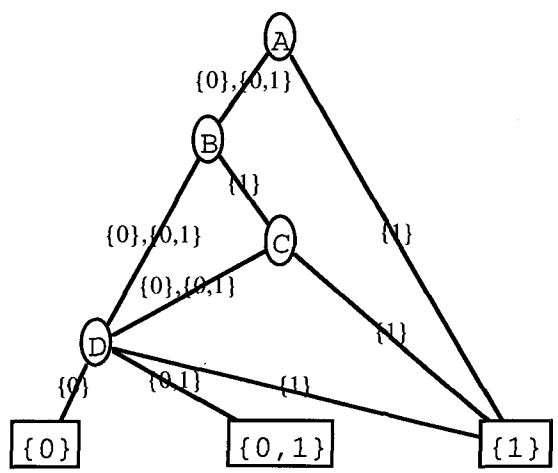


Figure 4. The DD for `A \/ B/\C \/ D`

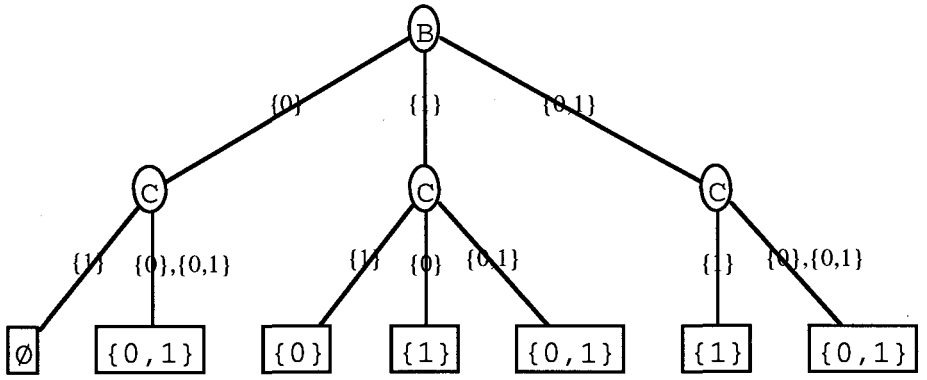
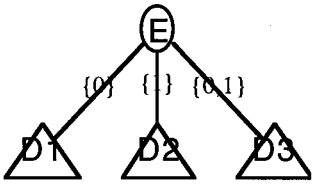


Figure 5. The template DD for C/\$_1$B

To define *pr* B, for a Boolean cterm B, we first convert B to a DD D and then convert D to *pr* D, a set valued expression. The translation from DDs to set expressions is trivial. If D is a terminal labeled Ø, {0}, {1}, or {0,1} the *pr* D is `{}`, `0`, `1`, or `0..1`, respectively. If D is a nonterminal of the form



then *pr* D is

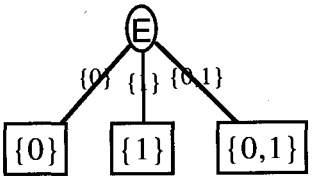
`b(`*pr* E, *pr* D1, *pr* D2, *pr* D3`)`.

If two of the branches go to the same DD, then they are replaced by the `b1` or `b2` branches as appropriate (see table 3 of appendix A).

The generalization of binary DD algorithms to four branch DDs is straightforward. To describe the algorithm, we use template DDs for each of the Boolean functions. For instance, the template DD corresponding to C/\$_1$B is shown in figure 5. This DD encodes all the usual propagation rules for a constraint such as `A /\ B #= C`. For instance, if `C:1` then Δ_A is set to {1} no matter what Δ_B is. If `B:1` and `C:0` then Δ_A is set to {0}. Finally, note that if `B:0` and `C:1` then the result is Ø, which causes backtracking.

A Boolean cterm B can be converted to a DD as follows. First, define an arbitrary order on the non-Boolean subterms. Next, replace all non-binary function symbols by equivalent forms using only binary function symbols. For instance ~B is replaced by B#`1`. Then, replace each 0 with a

terminal labeled by {0}, each 1 with a terminal labeled by {1}, and each non-Boolean subterm E (ie. variable or nested constraint) by the following DD:



Then, replace a subexpression of the form D_1 *b* D_2 where D_1 and D_2 are DDs and *b* is an binary Boolean function symbol with template DD $T(X_1,X_2)$. We describe how to replace such a subexpression by a DD next. To finish converting B we replace subexpressions of the form D_1 *b* D_2 until none remain.

To convert D_1 *b* D_2 using *b*'s template T(X,Y) to a DD D = *combine*$_T(D_1, D_2)$, use the following rules, which generalize the APPLY operation in [Brya92] to four branch DDs.

1. If D_1 and D_2 are both terminal then D is also terminal with

$$label(D) = T^*(label(D_1),label(D_2))$$

2. If one of D_1 and D_2 is nonterminal and the other is not, without loss of generality, assume D_1 is nonterminal. Then

$$label(D) = \quad label(D_1),$$

and for $B \in \{\emptyset,\{0\},\{1\},\{0,1\}\}$,

$$branch_D(B) = \quad combine_T(branch_{D_1}(B), D_2).$$

3. If both D_1 and D_2 are nonterminal and $label(D_1) \neq label(D_1)$ without loss of generality, assume $label(D_1)$ is before $label(D_1)$ in the cterm order. Then *label*(D) and *branch*$_D$ are the same as in rule 2 above.

4. If both D_1 and D_2 are nonterminal and $label(D_1) = label(D_1)$. Then

$$label(D) = \quad label(D_1),$$

and for $B \in \{\emptyset,\{0\},\{1\},\{0,1\}\}$,

$$branch_D(B) = \quad combine_T(branch_{D_1}(B), branch_{D_2}(B)).$$

For example, consider applying these rules with

$D_1(A,B) = A \ /\backslash \ B$ (figure 6),

$D_2(A) = \sim A$ (figure 7), and

$T(X,Y) = X \ \backslash/ \ Y$ (figure 8).

The result is

$$combine_T(D_1, D_2) = D(A,B) = (A \ /\backslash \ B \ \backslash/ \ \sim A) = (A \Rightarrow B) \text{ (figure 9).}$$

First, we apply rule 4 to nonterminals (1) of D_1 and (a) of D_1. This results in a nonterminal (1a) of with descendents defined by recursive combinations for each of the descendent DDs down corresponding branches. Next, we can apply rule 1 to terminals (2) and (b). Evaluating $T^*(\{0\},\{1\})$ gives $\{1\}$, the label of terminal (2b) in D. Rule 2 applies to nonterminal (3) and terminal (c), resulting in nonterminal (3c) with descendents defined by applications of rule 1 with terminals (4), (5), and (6) combined with (c), resulting in terminals (4c), (5c), and (6c), respectively. Finally, (7) combines with (d) according to rule 1 to produce (7d).

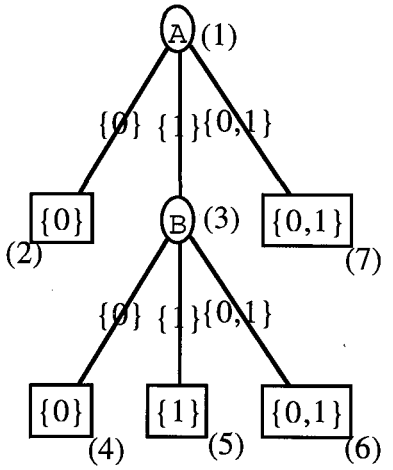


Figure 6. A DD for $D_1(A,B) = A \ /\backslash \ B$

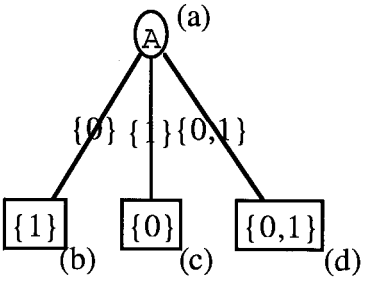


Figure 7. A DD for $D_2(A) = \sim A$

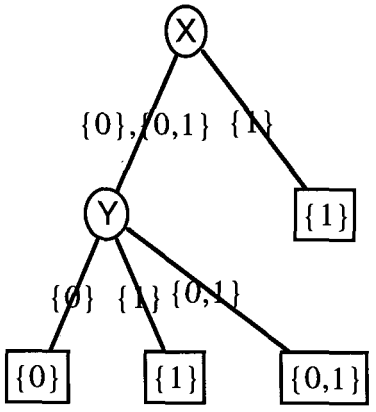


Figure 8. A template DD for T(X,Y) = X \/ Y

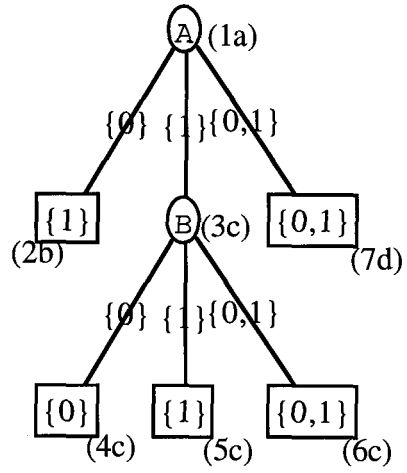


Figure 9. A DD for D(A,B) = (A /\ B \/ ~A) = (A => B)

4. Comparison with other CLP Languages

In this chapter, we compare Nicolog with most other CLP languages. Section 4.1 compares Nicolog with languages which process constraints using symbolic manipulation. Section 4.2 looks at the original domain manipulation based CLP languages and their successors. Section 4.3 explores relationships between Nicolog and domain manipulation based CLP languages with very similar capabilities.

4.1 Symbolic Manipulation Languages

As mentioned in the introduction, many CLP systems like CLP(R) [JM87], CAL [SA89], and Prolog III [Colm90] process constraints using symbolic manipulation algorithms. CHIP [DVS*88] also provides symbolic manipulations for linear rational number and Boolean domains. Symbolic manipulation algorithms for all but very limited domains tend to be very inefficient. For instance, the polynomial algorithms of CAL are doubly exponential ($\mathbf{O}(2^{2^n})$) in the worst case. The Boolean algorithms of CHIP, Prolog III, and CAL, though theoretically no less efficient than those of Nicolog (both $\mathbf{O}(2^n)$), are less efficient for many practical problems [DC93]. Evidence for this is in the popularity of a standard CHIP package which provides an alternative way to do Boolean constraint processing with the finite integer domain system by using the following equivalences:

```
~A ≡ 1-A

A /\ B ≡ A*B

A \/ B ≡ A+B-A*B
```

For efficiency, CLP(R), Prolog III and CHIP limit their real and rational domain constraint solving to only linear constraints and delay others until they become linear. This means these systems are far better than Nicolog for problems which can be formulated well with linear constraints. However, very little active constraint processing is possible for problems which are not linear. Boolean constraint processing can be formulated in linear systems as above, but again there is a loss in active constraint processing since the constraints are nonlinear. No major symbolic manipulation system allows mixing of Boolean and numeric constraints. All treat the constraint solver as a black box over which the programmer has no control and there is no way to program active constraints not supplied by the system.

4.2 The Original Domain CLP Languages and Their Successors

The original domain manipulation based CLP languages are CHIP [VanH89] and BNR Prolog [OV90]. A finite domain constraint system is supported by CHIP. It allows domains which are

small sets of integers or symbolic constants. Symbolic domain constraints are handled with a full arc consistency algorithm, while nonlinear integer constraints are handled with partial arc consistency algorithms very similar to Nicolog. Nicolog trigger tests are implemented as in the new CHIP compiler [AB91]. BNR Prolog also uses partial arc consistency algorithms for the real interval domain. BNR Prolog supports all of the usual real functions including exponential and trigonometric functions. Nicolog was designed to support the implementation of the BNR Prolog real constraint system. BNR Prolog decomposes complex constraints into basic constraints by introducing new variables. This is not necessary in Nicolog. CHIP and BNR Prolog were the first systems which allowed the user some control over the constraint solving procedure. Like Nicolog, both allow the user to write customized case analysis algorithms.

CHIP also has declarations which allow the user to implement nonprimitive constraints with arbitrary logic programs (LP constraints). When so declared, the LP is used to test combinations of values in an enumeration-based full arc consistency algorithm. For instance, consider the following LP constraint:

```
lookahead p(d,d).
p(X,Y) :- X+2 #=< Y ; Y+2 #=< X.
```

The `lookahead p(d,d)` declaration means that the constraint is triggered on any change to the domains of the variables. Other declarations allow LP constraints to be triggered when specified variables are instantiated. The above LP constraint is equivalent to

```
p(X,Y) :- X $= \ (<Y)-2..(>Y)+2, Y $= \ (<X)-2..(>X)+2.
```

in Nicolog. However, to achieve the same result, the LP constraint would enumerate the values in Δ_X and Δ_Y, removing values from one domain that fail to satisfy the predicate for any value from the other domain. Though this method of programing active constraints is very easy to use, it is not nearly as efficient. The PCs take constant time whereas the LP constraints take time proportional to the product of the domain sizes.

Aristo [EK92] improves LP constraints in two ways. The first improvement is lookahead declarations are replaced by trigger and satisfiability tests, as described in section 2.2.3. Instead of just any change or instantiation triggering a constraint as in CHIP, changes to the upper and lower bounds can also be specified. Nicolog also has this capability. The second improvement is instead of using the LP as a passive test in the arc consistency algorithm, Aristo accumulates domains which are the union of all solutions found for the LP constraint. This means Aristo only uses a partial arc consistency algorithm unless explicit enumeration is performed in the LP constraint. Thus, the LP constraint

```
p(X,Y) :- X+2 #=< Y ; Y+2 #=< X.
```

does the same thing as the Nicolog PCs above in constant time as well, though the PCs require less overhead and thus will be faster. However, if explicit enumeration is done in an LP constraint, Aristo gets the same effect as the full arc consistency based CHIP LP constraints. This is not available in Nicolog.

Echidna [HSS*92;SH92] handles the same finite domain constraints as CHIP and implements all the real numeric functions of BNR Prolog. However, instead of simple intervals, Echidna allows domains which are the union of disjoint intervals. This allows Echidna to handle real numeric constraints with discontinuous functions and disjunctive constraints, such as that defined by `p/2` above, more actively. This is because intervals can be deleted in the middle of real domains. Sidebottom and Havens [92] give an arc consistency algorithm called HACR. Like the PAC algorithm of figure 3, HACR needs to compute approximate projections of constraints. Thus Nicolog constraint compilation techniques and PCs can be applied directly in the implementation of HACR as well.

The original inspiration for Nicolog was CLP(BNR) [BO92], which evolved from BNR Prolog. As in Nicolog, arbitrary mixing of Boolean and numeric constraints is allowed in CLP(BNR). CLP(BNR) also handles both real and integer domains. However, CLP(BNR) has no way for the user to program custom nonprimitive constraints which can be programmed in Nicolog with PCs. Benhamou and Older [92] give a nice theory of how to compute projections for the various basic constraints used in CLP(BNR). However, they do not describe the details of how their theory is implemented, so it is hard to compare CLP(BNR) precisely with Nicolog. However, it appears that the Nicolog implementation of primitive constraints has the same propagation power as the equivalent constraints in CLP(BNR). Instead of DDs, CLP(BNR) implements Boolean constraints by using `min` for `/\`, `max` for `\/`, and `1-B` for `~B`.

4.3 CLP Languages Most Similar to Nicolog

The two systems most similar to Nicolog are cc(FD) [VHSD91; VHSD93] and clp(FD) [DC93; CD93]. The implementation of Nicolog is very similar to that of clp(FD) as described in [DC93].

Both clp(FD) and cc(FD) include the same subset of Nicolog PCs. The developers of those systems call PCs *indexical constraints* and use a slightly different syntax. The difference between Nicolog, cc(FD), and clp(FD) is that only Nicolog has conditional expressions and tests tests. However, cc(FD) and clp(FD) do include pointwise arithmetic functions on sets. Pointwise arithmetic, which could be added to Nicolog without difficulty, allows the implementation of full arc consistency. For instance, consider the constraint

```
X + 1 #= Y
```

An extremely active way of processing this constraint is if x is deleted from Δ_X then x+1 should be deleted from Δ_Y. Similarly, if y is deleted from Δ_Y then y-1 should be deleted from Δ_X. With its current compilatio scheme, Nicolog fails to do this if the deleted value is not a domain bound. In cc(FD) and clp(FD), we can program this behavior with the following PCs.

```
X $= dom(Y)-1,
Y $= dom(X)+1
```

where

$dom(Y)-1 = \{y-1 \mid y \in \Delta_Y\}$ and
$dom(X)+1 = \{x+1 \mid x \in \Delta_X\}$.

More generally, cc(FD) provides two distinct classes of primitive constraints: one which uses full arc consistency and one which uses interval based partial arc consistency. Full arc consistency is not currently available in Nicolog nor clp(FD). While cc(FD) uses an optimal arc consistency algorithm based on AC-4 [MH86; DVH91], Nicolog and clp(FD) use the simpler AC-3 algorithm [Mack77] augmented with triggers. Triggers appear to make AC-3 about as efficient as AC-4 in practice.

Boolean constraints can be implemented with PCs using only arithmetic [CD93] . For instance, for

```
A /\ B #= C,
```

Codognet and Diaz use

```
A $= (<C) .. (>C)*(>B)+1-(<B),
B $= (<C) .. (>C)*(>A)+1-(<A),
C $= (<A)*(<B) .. (>A)*(>B).
```

This handling is equivalent to the DD based PCs used by Nicolog. Arbitrary mixtures of Boolean and numeric constraints are not currently implemented in clp(FD). However, the PC language in clp(FD) can be extended with arbitrary functions programmed in C[15], so specific constraints can be implemented readily.

As well as PCs, cc(FD) has three general constraint classes: cardinality, constructive disjunction, and blocking implication. Each of these classes can be formulated in Nicolog with primitive constraints and/or PCs.

---

[15]Private communication with Daniel Diaz, September, 1993.

The general form of a cardinality constraint is

```
#(L,[C1, C2, ..., CN],H),
```

which means between L and H of the list of constraints `[C1, C2, ..., CN]` are true. This can be expressed straightforwardly in Nicolog by nesting constraints as follows:

```
C1 + C2 + ... + CN #= N, L #=< N, N #=< H.
```

Cardinality constraints are implemented by keeping true and false counters for the constraints `[C1, C2, ..., CN]`. The true (false) counter keeps track of the number of constraints which are true (false) for all values in their variable domains. If the true counter reaches `H`, the remaining constraints are forced false. If the false counter reaches `L`, the remaining constraints are forced true. This turns out to be exactly the effect of the equivalent Nicolog constraint. Cardinality constraints can be used to implement the Boolean constraints which are implemented with DDs in Nicolog. Arbitrary nesting of cardinality constraints is allowed. This makes arbitrary combinations of Boolean and numeric constraints possible. For instance,

```
(X #=< P) /\ (P #< X+S)
```

is equivalent to

```
#(2,[X #=< P, P #< X+S],2)
```

so

```
((X #=< P) /\ (P #< X+S)) #= B
```

is equivalent to

```
#(B,[#(2,[X #=< P, P #< X+S],2)],B).
```

Constructive disjunction is motivated by the need to do more active constraint processing than is possible with other implementations of disjunction. For instance,

```
max(A,B) #= C
```

can also be formulated by the predicate

```
max(A,B,C) :-
    A #=< C,
    B #=< C,
    (C #= A) \/ (C #= B).
```

Unfortunately, this gives the following behavior:

```
?- A:5..10, B:4..11, max(A,B,C).
A = _:5..10
B = _:4..11
C = _:5..inf
```

which is the same as cc(FD) if the disjunction is implemented with a cardinality constraint:

```
#(1, [(C #= A), (C #= B)], 2).
```

However, if we used the primitive maximum constraint, which can be compiled into PCs, we get more active constraint processing:

```
?- A:5..10, B:4..11, max(A,B) #= C.
A = _:5..10
B = _:4..11
C = _:5..11
```

which is the same result as cc(FD) if constructive disjunction is used in the predicate above. Constructive disjunction in cc(FD) is very powerful and has a fairly complicated implementation [VHSD93]. Nicolog PCs can implement the same behavior as simple cases of constructive disjunction which involve only interval reasoning.

Blocking implication constraints in cc(FD) are constraints like:

```
A #= B --> X #= Y.
```

Declaratively and operationally, this means the same as the Boolean implication:

```
A #= B => X #= Y.
```

In summary, Nicolog can implement a large part of cc(FD) with PCs alone. Major omissions from Nicolog are full arc consistency and more complex constructive disjunction constraints. However, in some ways, Nicolog is more flexible and extensible than than cc(FD). For instance, cc(FD) does not provide facilities for directly specifying constraint propagation rules such as those encoded in the square packing PCs of section 2.1.4. Moreover, Nicolog is conceptually simpler than cc(FD) and consequently easier to implement. This is demonstrated by the fact that Nicolog can be implemented with much less source code than cc(FD) [VHSD93].

5. Examples and Empirical Results

In this chapter, we give several examples which further demonstrate Nicolog's capabilities. We also compare a simple Nicolog implementation with some of the fastest and most powerful domain manipulation based CLP systems.

The Nicolog system consists of a compiler which translates Nicolog programs to instructions for our extension of the Warren abstract machine WAM [Warr83; Aït91] which we call the constraint logic abstract machine (CLAM). We use WAM extensions similar to those used by the CHIP compiler [AB91] and clp(FD) [DC93]. The Nicolog system is very simple, consisting of about 5500 lines of Prolog for the compiler and 5500 lines of C++ for the CLAM emulator, both including blank lines and comments. The extensions for constraints account for about half of the code in both the compiler and the emulator. The machine emulator was implemented rapidly, taking one person month to translate it from a Scheme prototype. The machine emulator code has not yet been profiled or optimized to any large extent. It was simply implemented in the way that seemed best from the outset given our experience with the Lisp prototype.

Only integer domains are currently implemented, but the emulator is configured to support real domains as well. Some intermediate computations (for example, those using division) are done using floating point numbers. Small domains, containing less than 32 consecutive integers, are implemented as a machine word and a base b where bit i is set iff the value $i+b$ is in the domain. Larger domains are implemented as intervals with pairs of bounds. Thus, disequality constraints and PCs using the set complement operator sometimes do nothing to large domains when they should be able to punch holes in the middle of intervals.

All computation results in this chapter are given using a Nicolog system running on a Sun Sparcstation IPX which runs at 28.5 Mips. Comparative results for other systems are normalized to eliminate machine speed differences.

5.1 Cryptarithmetic

In this section, we see that for cryptarithmetic problems, it can be more efficient to avoid decomposing complex constraints into sets of basic constraints by introducting new variables. A cryptarithmetic problem is of the following form: given a word equation such as SEND + MORE = MONEY, find an assignment of digits to letters which makes the equation true. Each letter must be assigned a different digit and no resulting number can start with the digit 0. The cryptarithmetic problem is a good test to see if Nicolog's method of handling complex constraints can be more efficient than decomposing complex constraints into basic constraints with extra variables. In Nicolog, SEND + MORE = MONEY can be naturally encoded as:

```
sendmory(S,E,N,D,M,O,R,Y) :-
    [S,E,N,D,M,O,R,Y]:0..9,
    allDiff([S,E,N,D,M,O,R,Y]), S #/= 0, M #/= 0,
              1000*S + 100*E + 10*N + D +
              1000*M + 100*O + 10*R + E #=
    10000*M + 1000*O + 100*N + 10*E + Y,
    label([S,E,N,D,M,O,R,Y]).
```

`AllDiff` sets up a disequality constraint between the argument variables, forcing them to all have different values. `Label` implements a backtrack search (with interleaved calls to the consistency algorithm) for values for the given variables. From the complex constraint, Nicolog produces eight independent isolations, such as:

```
S#=(-(D)-91*E+9000*M+90*N+900*O-10*R+Y)/1000
```

This isolation corresponds to the following optimized PC:

```
S$= (-(>D)-91*(>E)+9000*(<M)+90*(<N)+900*(<O)-10*(>R)+(<Y))/1000..
    (-(<D)-91*(<E)+9000*(>M)+90*(>N)+900*(>O)-10*(<R)+(>Y))/1000
```

Using these PCs, the PAC algorithm is identical to the consistency algorithm for linear integer arithmetic used in CHIP [VanH89]. For this complex constraint involving 8 variables with 10 possible values each, arc revision takes time proportional to the number of variables. Enumeration-based arc revision like that used in AC-3 is not practical since there are around 10^8 combinations to enumerate. Moreover, most of the work done by enumeration-based arc revision is useless since almost all the domain refinements can be calculated with only domain bounds.

To compare the Nicolog approach with systems that decompose constraints, we re-expressed the above constraint with a large number of constraints of the form `A + B #= C` and *n*`*A #= B` where *n* is a number. Note that not only are decomposed constraints more expensive because of the extra constraints and variables, but they are also weaker in search pruning power because the consistency algorithm can only use part of the whole constraint at any one arc revision.

To make the comparison as fair as possible, we implemented special instructions in the Nicolog abstract machine to process these two basic constraints as efficiently as possible. Processing the constraint directly took 17.66 ms to complete the search for all solutions (finding only one) and the decomposed constraint processing took 26.53 ms. So for this program, processing complex constraints directly is about 1.5 times faster.

5.2 N-Queens

N-Queens is the problem of placing n queens on an n by n chess board so that no two queens attack each other. It has been widely used to demonstrate the capabilities of various constraint processing algorithms. In this section, we experiment with various ways of handling the 'no attack' constraint, showing some of the flexibility of Nicolog.

A now classic CLP solution to the problem is given in [VanH89]. The problem is formulated with a variable Q_i with domain `1..n` for each queen. If a solution has $Q_i = j$, this means that a queen is placed in rank i, file j, on the chess board. In Nicolog, this program is written as follows:

```
queens(N, Qs) :-
    length(Qs,N),
    Qs:1..N,
    safe(Qs),
    label(Qs).

safe([]).
safe([Q|Qs]) :-
    noattack(Qs,Q,1),
    safe(Qs).

noattack([],_Q,_N).
noattack([QN|Qs],Q,N) :-
    QN #/= Q,
    QN #/= Q - N,
    QN #/= Q + N,
    N1 is N + 1,
    noattack(Qs,Q,N1).
```

As described in [VH89], if `label` instantiates the variables in order, the forward checking (FC) algorithm is used. If `label` uses `deleteff` to instantiate variables in the dynamic order of increasing domain size, then forward checking with the first fail principle [HE80] (FCFF) is implemented. Since we wish to compare this program with others, we label the FC version FCorig.

With Nicolog's rich constraint language, we can easily try many different approaches to constraint processing. For instance, the number of constraints per pair of queens can be reduced from 3 to 2 using the absolute value function in noattack, as follows:

```
noattack([QN|Qs],Q,N) :-
    QN #/= Q,
    abs(QN-Q) #/= N,
    N1 is N + 1,
    noattack(Qs,Q,N1).
```

We will call the FC program with this clause FCabs.

In Nicolog, we can maximize the efficiency of constraint processing by using PCs instead of primitive constraints. Observe that if a queen variable `Q` becomes instantiated, then we can delete the values `Q`, `Q-N`, and `Q+N` from the domain of another queen variable which is N elements away in the list of all queens. Thus, we can rewrite `noattack` with just two PCs as follows:

```
noattack([QN|Qs],Q,N) :-
    Q $= \{QN, QN+N, QN-N},
```

```
    QN $= \{Q, Q+N, Q-N},
    N1 is N + 1,
    noattack(Qs,Q,N1).
```

These PCs combine the six PC which result from the compilation of the disequalities into just two PCs. The two PCs are symmetrical, so we will explain only the first:

```
Q $= \{QN, QN+N, QN-N}
```

Since QN appears in an expression, this PC is not triggered until QN is instantiated to a constant. So operationally, it means when QN is ground, delete QN, QN+N, and QN-N from Δ_Q. This is exactly what the three disequalities would do with three different PCs. We call the FC and FCFF programs using this clause FCcust and FCFFcust, respectively.

This custom programmed constraint is more efficient than those above for two reasons. First, it does not attach any projection atoms to the variable N, which we know is a constant at run time. Of course, a smart compiler could deduce this automatically since N is in a call to `is`. Second and more importantly, the custom constraint combines the three constraints into one, and they all use a common groundness test instead of testing the groundness of queen variables independently.

As efficient as the custom constraint is, it can further be improved when we are using the FC algorithm. Since the variables are instantiated in the order given by the list of queen variables, arc revisions on arcs which point backward in the list are useless. This is because they only check what the converse forward arc has already ensured. Thus, we would be better off doing only directed arc consistency by using only arcs which point forward in the list. We can do this by deleting the backward PC, as shown in the following `noattack` clause:

```
noattack([QN|Qs],Q,N) :-
    QN $= \{Q, Q+N, Q-N},
    N1 is N + 1,
    noattack(Qs,Q,N1).
```

We call the FC and FCFF programs using this clause FCcustdir and FCFFcustdir, respectively. Directed arc consistency is also important because it can be used to efficiently solve CSPs with tree like structures [Freu82; Freu85; DP89].

The following table gives the time in seconds to find the first solution[16] for the four constraint implementations above using the FC algorithm:

---

[16]We do not use time to find all solutions because, for larger *n*, there are many, many solutions.

n	FCorig	FCabs	FCcust	FCcustdir
8	0.15	0.27	0.16	0.17
10	0.18	0.47	0.16	0.16
12	0.48	1.637	0.23	0.18
14	3.38	13.50	0.58	0.35
16	19.75	85.53	3.52	1.88

The table shows the time in seconds for Nicolog to find the first solution. From these results, we can see custom constraints are by far more efficient than the others, and the directed constraint is much faster than the non-directed one as n grows. It is also clear that the complexity of processing constraints involving absolute value outweighs the benefit of reducing the number of constraints they provide.

As n grows large, Van Hentenryck [89] showed that the FCFF algorithm is far superior to the FF algorithm. So we would expect that the FCFF algorithm would beat an FC algorithm if n were large enough. Note that FCFF instantiates at least some of the variables in the order given by the list. So can directed constraints be used to improve the FCFF algorithm? The following table compares time in seconds to find the first solution using the two possibilities:

n	FCFFcust	FCFFcustdir
20	0.15	0.53
22	0.50	0.23
24	0.10	1.83
26	0.35	0.53
28	0.10	2.52
30	0.25	27.17
32	0.22	1.47

So it is clear that as n grows, the pruning performed by the backward arcs is critical to the efficiency of the FCFF algorithm.

5.3 The Schur Lemma: a Classic Boolean Benchmark

In order to see how Nicolog's use of DDs for Boolean constraints compares with other systems, we have compiled execution times of a program for the so called Schur lemma problem [BO92; CD93]. The problem is to try to put n balls labelled by the integers $\{1,\ldots,n\}$ into three boxes so

that for any triple (x,y,z) such that $x + y = z$, balls x, y, and z are not all in the same box. This problem has a solution iff $n < 14$.

To benchmark Boolean systems, this problem is formulated as a Boolean matrix M_{ij} $(1 \le i \le n, 1 \le j \le 3)$ where M_{ij} is true iff ball i is in box j. The constraints for this problem are each ball must be in exactly one box:

$$M_{i1} \; + \; M_{i2} + \; M_{i3} \; \#= \; 1 \; (1 \le i \le n)$$

and for each (x,y,z) such that $x + y = z$ and j

[23] $$\sim (M_{xj} \; /\backslash \; M_{yj} \; /\backslash \; M_{zj}) \; (1 \le j \le 3)$$

for the requirement that these balls are not all in the same box.

The best results for this problem are obtained by clp(FD) [CD93], which is about 5 times faster than CHIP with this formulation for n = 13 and 14, and about 12 times as fast for n = 30. The following table compares more recent clp(FD) results with Nicolog both running the same program (given in appendix C) on a Sun Sparcstation with a 28.5 Mips processor. The times to find all solutions are in seconds.

n	Nicolog	clp(FD)	$\frac{\text{Nicolog}}{\text{clp(FD)}}$
13	0.83	0.092	9.0
14	0.85	0.093	9.1
30	2.53	0.25	10.1
100	11.21	1.27	8.8
200	26.15	3.47	7.5
300	48.04	6.57	7.3
400	72.84	10.54	6.9
500	101.67	15.41	6.5

As n grows, Nicolog improves relative to clp(FD). Since the same program is run in both systems, the only explanation for the improvement can be differences in how constraints are handled. Two differences may contribute to the improvement. First, since clp(FD) does not have DDs for Boolean constraints, clp(FD) formulates [23] with arithmetic PCs

$$M_{xj} \; \$= \; 0..1 - \; <M_{yj} \; * \; <M_{zj} \, .$$

Note that this PC sets M_{xj} to 0 if M_{yj} and M_{zj} both become 1. This is exactly the same behavior of the DD based PCs Nicolog compiles from [23]. To see how much difference this makes, we can compare the data for the DD based Nicolog constraints in the table above with arithmetic based constraints in Nicolog. The following table compares the ratio of the arithmetic handling over the DD handling.

n	arith/DD
100	1.23
200	1.23
300	1.17
400	1.17
500	1.15

Since the advantage of using DDs decreases as n increases, we can assume that using DDs is not a major factor in Nicolog's improvement.

The correct explanation for Nicolog's improvement is that clp(FD) spends an increasingly larger proportion of its execution time generating the CSP before the searching part of the program is executed. The following table shows the percentage of total execution time spend by Nicolog and clp(FD) constructing the CSP before starting the search.

n	Nicolog	clp(FD)
100	12	42
200	21	59
300	31	71
400	39	77
500	47	82

So clp(FD) is very fast at searching, but it spends most of its time constructing the CSP as n increases. The most likely explanation for this difference is that the Nicolog abstract machine adds a constraint to a CSP with a single complex instruction while clp(FD) does the same with several simple instructions. This difference appears to be an important factor as n grows large.

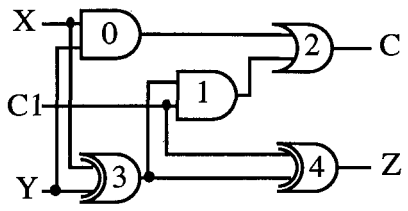


Figure 10 A full adder

5.4 Digital Circuit Diagnosis

The digital circuit diagnosis problem is as follows.

Given:

1. A description of a digital circuit with a set of components *C*.
2. A function *f* computed by the circuit.
3. A symptom consisting of an input output pair (*in*,*out*) such that $f(in) \neq out$.

Find:

A diagnosis $D \subseteq C$ which, if not working correctly, could result in the circuit computing *out* given *in*.

The motivation behind the following example is twofold. First, the diagnosis of digital circuits is an interesting problem which needs to be solved in the real world. Second, traditional Boolean benchmarks are not very good indications of the general usefulness of Boolean constraint solvers. For instance, the Boolean formulations of the Schur lemma, *n*-queens, and pigeon hole principle [BO92; CD93] have very simple clausal forms and are not difficult to solve. However, many interesting Boolean problems have no simple transformation to clausal form. Section 5.3 shows that Nicolog performs well on problems with simple formulations in terms of the basic connectives: `/\`, `\/`, and `~`. Here, we are interested in showing Nicolog's capabilities for a problem which has no simple basic formulation. We aim to show that a very short, simple and clear Nicolog program solves the problem very efficiently.

The specific circuit we will be using for benchmarks is an *n* bit adder with forward carry propagation. However, any combinatorial circuit diagnosis problem could easily be formulated from its network description. An *n* bit adder is constructed from *n* full adders where a full adder circuit is shown in figure 10. For bit *i*, ($0 \leq i < n$) $\mathtt{X}_i$ and $\mathtt{Y}_i$ are the input bits, $\mathtt{Z}_i$ is the output bit, $\mathtt{C1}_i$ is the input carry bit and $\mathtt{C}_i$ is the output carry bit. $\mathtt{C1}_0$ is the input carry to the *n* bit adder and

C_{n-1} is its output carry. Forward carry propagation is achieved by connecting C_i to $C1_{i+1}$ ($0 \leq i \leq n$-2). Let num_n be a function for converting from n element lists of bits to numbers defined by:

$$num_n([X_0,\ldots,X_i,\ldots,X_{n-1}]) = X_0 \cdot 2^0 + \ldots + X_i \cdot 2^i + \ldots + X_{n-1} \cdot 2^{n-1}$$

Then an n bit adder computes the function:

$$f([X_0,\ldots,X_{n-1}],[Y_0,\ldots,Y_{n-1}],C1) = ([Z_0,\ldots,Z_{n-1}],C)$$

where

$$num_n([X_0,\ldots,X_{n-1}]) + num_n([Y_0,\ldots,Y_{n-1}]) + C1 = num_n([Z_0,\ldots,Z_{n-1},C])$$

This example was constructed based on an example in [Colm90]. The core of the program is the following predicate:

```
fullAdder(X, Y, C1, Z, C, D0, D1, D2, D3, D4) :-
    ~D0 => (U1 <=> X /\ Y),
    ~D1 => (U2 <=> U3 /\ C1),
    ~D2 => (C <=> U1 \/ U2),
    ~D3 => (U3 <=> X # Y),
    ~D4 => (Z <=> U3 # C1).
```

This predicate specifies a full adder as depicted in figure 10 with input wires `X`, `Y`, and `C1`; output wires `Z` and `C`; and internal wires `U1`, `U2`, and `U3`. Thus, the right hand side of each implication specifies the relationship between the wires of a given gate. The full adder has 5 components labeled 0 to 4 in figure 10. For each component j we have a Boolean variable `D`j which we interpret as true iff component j is faulty. Thus, each of the implications means if a component is not faulty then it enforces the proper relation between its wires. Moreover, the set of `D` variables which are true define the set of components which form a diagnosis for the circuit. It should be noted that each of the constraints in the fullAdder predicate has a very complicated clausal form. Even their negation normal forms are reasonably complicated.

In order to construct an n bit adder, we need n element lists `Xs`, `Ys`, and `Zs` of Boolean variables, as well as `C1` and `C`, the initial input and output carries. We index lists from the left starting at 0. The call

```
?- adder(Xs,Ys,Zs,C1,C,Ds,[]).
```

will string together n full adders to create an n bit adder with the given inputs and outputs. `Ds` is a $5n$ element list of Boolean diagnosis variables where the jth element of D is true iff the component c in the adder for bit i is faulty where $j = i \cdot 5 + c$ and $0 \leq c < 5$. That is i is the quotient of j and 5 and c is the remainder.

The adder predicate is defined by the following two clauses:

```
adder([],[],[],C,C,Ds,Ds).
adder([X|Xs],[Y|Ys],[Z|Zs],C1,C,[D0,D1,D2,D3,D4|Ds1],Ds) :-
    fullAdder(X,Y,C1,Z,C2,D0,D1,D2,D3,D4),
    adder(Xs,Ys,Zs,C2,C,Ds1,Ds).
```

So we can input fault symptoms as atomic numbers instead of binary lists, we define a predicate `bits(N,X,Xs)` which is the relation $num_N(Xs) = X$. It is defined as follows:

```
bits(N,X,Xs) :-
    length(Xs,N),
    Xs:0..1,
    bits1(Xs,0,N,X).

bits1([],N,N,0).
bits1([Xi|Xs1],I,N,X) :-
    I < N,
    I1 is I + 1,
    X #= Xi*2^I+X1,
    bits1(Xs1,I1,N,X1).
```

`Bits/3` can be used to compute both num_n and its inverse.

Now, we need a predicate to put the previous predicates together:

```
nadder(N,X,Y,Z,C1,C,Ds) :-
    [X,Y,Z]:0..2^N-1,
    [C1,C]:0..1,
    bits(N,X,Xs),
    bits(N,Y,Ys),
    bits(N,Z,Zs),
    adder(Xs,Ys,Zs,C1,C,Ds,[]).
```

`Nadder(N,X,Y,Z,C1,C,Ds)` creates an `N` bit adder with the given symptom inputs `X`, `Y`, `C1` and outputs `Z`, `C` and produces the diagnosis tuple `Ds`. Interestingly, it can also be run backwards to produces symptoms given diagnoses.

Now `nadder/7` alone is not very useful, since one diagnosis for any symptom is that every component is faulty. However, many components breaking simultaneously is very unlikely. What we are really interested in is minimal diagnoses. We can use the following predicates to help us find them:

```
diagnose(N,X,Y,Z,C1,C,Ds,F) :-
    F:0..N*5,
    nadder(N,X,Y,Z,C1,C,Ds),
    X+Y+C1 #/= Z+C*2^N,
    sum(Ds,F).
```

```
sum([],0).
sum([X|Xs],S) :-
    S #= X + S1,
    sum(Xs,S1).
```

The predicate diagnose adds an extra variable F which is the number of faults, as well as enforcing the *f(in)* ≠ *out* constraint from the symptom specification. Note that counting the number of true Boolean variables is very complicated using the basic Boolean connectives. For instance, suppose we want to state that exactly F elements of an *m* element list Ds are true. One way to do this is with a disjunction

$$D_{p_1} \backslash/ \ \ldots \ \backslash/ \ D_{p_F}$$

for each subset of size F and a negated conjunction

$$\sim(D_{q_1} \ /\backslash \ \ldots \ /\backslash \ D_{q_{F+1}})$$

for each subset of size F+1. So the number of constraints grows exponentially with the number of faults. However, in order to do minimal fault diagnosis, we take F as a variable and minimize it. It is not clear that there is any good way to express this using the usual Boolean connectives. Thus, pure Boolean logic would probably have to iteratively increase F, attempting a new and bigger Boolean problem as F increases. Systems which allow summing of Boolean variables can set up the Boolean problem only once and then search for solutions which minimize F.

Now, we have a fairly powerful system for diagnosing full adders. Here are some examples of its capabilities (`label(Ds)` searches for values for the variables in Ds using backtrack search):

```
?- diagnose(2,0,0,2,1,0,Ds,1), label(Ds).
Ds = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
Alternatives? (y/n) y
No more solutions
```

So if a 2 bit adder gives the symptom 0+0+1=2+0, then the only possible diagnosis assuming exactly 1 fault is component 3 in bit 0.

As mentioned earlier, we can also run the program backwards to generate symptoms from diagnoses:

```
?- diagnose(2,X,Y,Z,C1,C,[1,0,0,0,0,0,0,0,0,0],1),
   label([X,Y,Z,C1,C]).
C = 0
C1 = 0
Z = 2
Y = 0
X = 0
```

```
Alternatives? (y/n) y
C = 0
C1 = 0
Z = 3
Y = 1
X = 0
...
```

For a larger example, consider the following 27 bit adder query:

```
?- diagnose(27,0,134217727,134217727,1,1,Ds,1), label(Ds).
No solutions
```

This means that no single fault diagnosis can explain this symptom. We can try again with exactly two faults:

```
?- diagnose(27,0,134217727,134217727,1,1,Ds,2), label(Ds).
D = [...] % bit 0 component 3 and bit 26 component 2 are broken
Alternatives? (y/n) y
D = [...] % bit 0 component 3 and bit 26 component 1 are broken
Alternatives? (y/n) y
D = [...] % bit 0 component 3 and bit 26 component 0 are broken
Alternatives? (y/n) y
No more solutions
```

So in this example, assuming two faults, bit 0 component 3 is definitely broken and one of components 0, 1, or 3 in bit 26 are broken.

We can also minimize the number of faults in the diagnosis as follows:

```
?- diagnose(27,134217727,134217727,134217727,0,0,Ds,F),
   label([F|Ds]).
F = 2
Ds = [...] % bit 0 component 5 and bit 26 component 3 are broken
Alternatives? (y/n) n
```

Since `label/1` tries values in increasing order, the first solution will have the minimal number of faults.

Nicolog answers these queries almost instantly. In fact, it is hard to find an example where more than one or two faults are required for diagnosis and it is hard to make Nicolog take any significant time to answer these queries. However, at least one class of symptoms requires at least n faults to diagnose in an n bit adder:

$X = 0$

$Y = 0$

$Z = 2^n-1$

C1 = 0

C = 0

Though the probability of this class of failures is almost zero, it makes a good benchmark for Boolean systems. Once again, we compare with clp(FD) [CD93;DC93], one of the fastest CLP languages that can mix Boolean and numeric constraints. The following table shows the time to find the first minimal fault solution in seconds.

n	Nicolog	clp(FD)	$\frac{\text{Nicolog}}{\text{clp(FD)}}$
1	0.16	0.004	40.0
2	0.17	0.008	21.3
3	0.66	0.020	33.0
4	0.27	0.091	2.96
5	1.75	0.545	3.21
6	11.83	3.787	3.12
7	74.66	27.547	2.71
8	522.40	175.118	2.98

For sufficiently large n , it appears that clp(FD) is about three times faster than Nicolog for this problem. We note that the clp(FD) program is identical to the Nicolog program except for the way some constraints are handled. Since clp(FD) does not currently support DDs, it uses PCs that simulate the effect of DDs using arithmetic. For instance, for a constraint such as[17]:

```
X # Y #= Z,
```

clp(FD) uses 3 PCs of the form

```
Z $= (X+Y) mod 2.
```

This PC is not triggered until X and Y are both instantiated to Boolean constants. Then, Z is set true if X and Y are different and false otherwise. This is exactly the same behavior as the DD based PC compiled from the Nicolog program.

The Schur lemma benchmark for $n = 500$ generates a far larger CSP than this diagnosis benchmark for $n = 8$. This leads us to believe that the CSP generation efficiency is not a major factor the diagnosis benchmark. Since clp(FD) is at least six times faster than Nicolog on the Schur lemma

[17]Recall that # means 'exclusive or.'

benchmark for $n = 500$ while only about three times as fast for this diagnosis benchmark for many small instances, it appears that DD based PCs are more efficient for hard problems with complex Boolean constraints.

5.5 The Magic Sequence Problem

The magic Sequence problem has been used frequently to test various CLP languages [VanH89; VD91; AB91; BO92; VHSD93]. The magic sequence problem is: find a sequence of n nonnegative integers $[x_0,...,x_{n-1}]$ such that for all i $(0 \le i < n)$, x_i is the number of occurrences of the integer i in the sequence. In other words, for all i $(0 \le i < n)$,

[24] $$x_i = |\{i \mid x_j = i \ (1 \le j \le n)\}|$$

It turns out that there is only 0, 1, or 2 magic sequences for a given n. Here are some magic sequences:

n	magic sequences
1,2,3	none
4	[1, 2, 1, 0], [2, 0, 2, 0]
5	[2, 1, 2, 0, 0]
6	none
7	[3, 2, 1, 1, 0, 0, 0]
8	[4, 2, 1, 0, 1, 0, 0, 0]
9	[5, 2, 1, 0, 0, 1, 0, 0, 0]

For $n \ge 7$, there is a single solution of the form $[n\text{-}4,2,1,0^{n-7},1,0,0,0]$. As equation [24] suggests, this problem is naturally formulated in terms of cardinality constraints. Van Hentenryck and Deville [91] show that using cardinality constraints is far faster than the usual CLP approach to this problem [VanH89]. In Nicolog, we can naturally express cardinality constraints as sums of nested equality constraints. Thus, we can write a very elegant Nicolog program to solve the problem:

```
magic(N,Xs) :-
  length(Xs,N),
  Xs:0..N,
  constrs(Xs,0,Xs),
  labelff(Xs).

constrs([],_N,_Xs).
constrs([XI|Xs1],I,
        Xs) :-
  sum(Xs,I,XI),
  I1 is I + 1,
  constrs(Xs1,I1,Xs).

sum([],_I,0).
sum([XJ|Xs],I,S) :-
  S #= (XJ#=I) + S1,
  sum(Xs,I,S1).
```

`Labelff(Xs)` does a backtrack search for values for `Xs` using the first fail principle [VanH89]. This means that at each level in the search, a variable with the smallest domain is selected to be instantiated next. The call `sum(Xs,I,XI)` is equivalent to equation [24] above. The call

`consts(Xs,0,Xs)` makes equation [24] true for each element `XI` of the list `Xs`. Benhamou and Older [92] show that this problem can be solved much faster if the following redundant constraints are added:

$$\sum_{0 \le i < n} x_i = n \qquad\qquad \sum_{0 \le i < n} i\,x_i = n$$

These redundant constraints are easy to program in Nicolog using the following two predicates

```
sum1([],0).
sum1([X|Xs],S) :-
    S #= X + S1,
    sum1(Xs,S1).

sum2([],_I,0).
sum2([XI|Xs],I,S) :-
    S #= XI*I + S1,
    I1 is I + 1
    sum2(Xs,S1).
```

Here are the times in seconds for Nicolog to find all solutions using the program above with the two redundant constraints added:

n	time (s)	*n*	time (s)
10	0.23	60	21.70
20	1.13	70	32.48
30	3.38	80	46.87
40	7.42	90	64.95
50	13.32	100	87.21

Two of the fastest systems for solving this problem are clp(FD) [DC93] and cc(FD) [VHSD93]. A clp(FD) program, which processes the constraints in exactly the same way as Nicolog, runs about seven times faster on average. The results in [VHSD93] are for a program which uses cardinality constraints instead of summing constraints. For the data given in [VHSD93] (n = 12,17,22), cc(FD) is a little over twice faster than Nicolog.

5.6 Disjunctive Scheduling

The bridge scheduling problem described in [VanH89] has been widely used to benchmark CLP systems. This problem involves scheduling 45 fixed duration tasks in a way that minimizes the completion time of the project. The problem involves precedence, distance, and disjunctive constraints. A task i can be formulated by a variable start time S_i and a fixed duration D_i. The initial domains for the start time variables is 0 to the sum of all task durations. A precedence

constraint between task i and task j means that i must be finished before j starts. This can be formulated by the constraint

$$S_i+D_i \ \#=< \ S_j.$$

Distance constraints, such as i must end no later than five days after j starts, can be formulated with constraints like

$$S_i+D_i \ \#=< \ S_j+5.$$

Disjunctive constraints, which result from tasks that must use the same resource exclusively, can be formulated as the disjunction of precedence constraints.

Van Hentenryck [89] observed that interval based constraint propagation is sufficient to solve scheduling problems with only precedence and distance constraints without any search at all. He shows that the classical critical path method (CPM) algorithm is identical to applying interval based arc consistency, assigning the task completion time to the minimum value in its domain, and then applying interval based arc consistency once again. After this, each task with a single possible value is on the critical path and all other tasks have slack defined by the range of their domains.

Scheduling with disjunctive constraints is *NP*-complete, so the real challenge is how to handle disjunctive constraints. Van Hentenryck [89] showed that, since arc consistency solves the problem without disjunctive constraints, a good approach is to search for an ordering of the disjunctive tasks which minimizes the completion time using a branch and bound algorithm. The branch and bound is implemented by a predicate which searches for a solution, asserts its completion time as the earliest completion time found so far and then starts the search again with a new constraint that the completion time has to be earlier. This process continues until no new solution is found. Then, the completion time for the last solution is optimal.

The solution Van Hentenryck gave formulated disjunctive constraints with a nondeterministic predicate:

```
disj(S1,D1,S2,D2) :- S1+D1#=<S2.
disj(S1,D1,S2,D2) :- S2+D2#=<S1.
```

This means that only one of the inequalities can be active at a time. However, it is possible to use the constraints more actively. Earlier choices can cause one of the inequalities to become false (true), forcing the other to be true (false). Frederick Benhamou[18] provided a program which uses

---

[18]private communication, May 1993.

nested inequalities to get more active constraint propagation by using the following predicate for disjunctive tasks:

```
disj1(S1,D1,S2,D2,B) :- (S1+D1#=<S2) #= B, B \/ (S2+D2#=<S1).
```

In this predicate `B` is true exactly when task 1 precedes task 2 and false exactly when 2 precedes 1. With this predicate, all the disjunctive constraints are active before any searching takes place. The search for an ordering of disjunctive tasks is achieved by collecting a list `Bs` of Boolean variables, one for the fifth argument of each call to `disj1/5` and then searching for values for each variable in `Bs`.

We timed Nicolog running a program supplied by Benahmou, which is given appendix D. This program implements disjunctive constraints using the `disj1/5` predicate given above. For this program, Nicolog takes 2.98 seconds total time to find and prove the optimal solution. In [AB91], it is stated that the CHIP compiler system takes 2.2 seconds to do the same. Though they do not give their program, we presume that it uses the `disj/4` predicate above. In [VHSD93], a time of 2.88 seconds is reported for cc(FD). The indicate that they handle disjunctive constraints in a way similar to `disj1/5` above, except they use cardinality constraints instead of embedding constraints in a disjunction.

5.7 Square Packing

The time in seconds to find the first solution with the square packing program given in section 2.1.3 are as follows.

Problem	time (s)
1	1.47
2	2.60
3	61.58
4	94.92

The cc(FD) program in [VHSD93], from which our program was derived, is identical except cardinality constraints are used instead of Boolean constraints. Using this program, cc(FD) solves problem 3 in 37.9 seconds.

The optimizations using PCs given in section 2.1.5 involve the set complement operator. Thus, they are only effective when domains are represented in a way that allows deletion of arbitrary elements. In Nicolog, this means only domains with less than 32 elements, since larger domains are represented by intervals. So currently, only the solution time of problem 1 can be significantly

decreased with these optimizations. Optimizing both `noOverlap2/6` and `sumOfSqsWith/4` with PCs, problem 1 is solved in 1.00 seconds. If the same speedup resulted for problem 3, Nicolog could solve it in 41.9 seconds, almost as fast as cc(FD).

5.8 Summary of Results

In this chapter, we have shown that the Nicolog approach to CLP is viable. The examples here show that the Nicolog compiler can usually automatically generate PCs which implement the same constraint propagation algorithms used by other systems. We have also shown that PCs make it possible to implement optimized constraint propagation algorithms for complex constraints. Nicolog, though it is simple unoptimized research software, is able to solve hard problems in time comparable to the most efficient CLP systems.

6. Conclusions and Future Work

In this thesis, we described Nicolog, a domain based CLP language which is suited to the development of hybrid consistency/case analysis algorithms. The main observation exploited in this thesis is that the approximate projection of mathematical relations is a key operation in arc consistency algorithms suitable for combination with case analysis algorithms such as backtrack searching. In order to take advantage of the importance of approximate projection in arc consistency, we defined an new class of constraints called projection constraints (PCs). PCs encapsulate the knowledge of how to do approximate projection, opening up the arc consistency algorithm to Nicolog programmers. PCs allow programmers to fine tune and extend the capabilities of an arc consistency algorithm. Moreover, they are well suited as a target language for the compilation of many classes of constraints, including mixed Boolean/numeric, non-linear, cardinality, constructive disjunction, and implication constraints [VHSD93]. In fact, PCs can be seen as a reduced instruction set suitable for implementing most forms of local constraint reasoning. We gave and formally verified the translation scheme used by Nicolog to compile a very general class of constraints available in CLP(BNR) [BO92] into PCs. CLP(BNR) constraints include non-linear numeric, Boolean, as well as arbitrary mixtures with nested constraints.

We showed how the interval reasoning and case analysis supported by PCs can be used to avoid inefficient enumeration-based arc revision. We also showed how complex constraints can be handled directly, instead of decomposing them to semantically equivalent but sometimes less efficient sets of basic constraints. We have seen that PCs can also be used to efficiently implement most of the constraint reasoning capabilities available in other domain manipulation based CLP languages.

We gave several short Nicolog programs which solved complex and fairly difficult problems. We also observed that a very simple implementation of Nicolog runs with speed comparable to some of the fastest CLP systems available. In particular, Nicolog is about as fast as CLP(BNR) [BO92] and CHIP [VanH89], but much more flexible and extensible. Nicolog is also more flexible and extensible than clp(FD) [DC93], which only implements the subset of Nicolog PCs that are also found in cc(FD) [VHSD93][19]. Though Nicolog is substantially slower than clp(FD), there are only superficial differences between the extensions of the WAM used to implement Nicolog and clp(FD). Thus, the difference in performance is primarily due to the fact that clp(FD) compiles WAM instructions into C code instead of using a software emulator as Nicolog does. cc(FD) is the only CLP language with flexibility and extensibility comparable to Nicolog. As well as a subset of

[19]Recall that in cc(FD) and clp(FD), PCs are called indexical constraints.

PCs, cc(FD) has cardinality, constructive disjunction, and blocking implication constraints which are not available in Nicolog. However, Nicolog can solve many problems that require the extra constraints in cc(FD) with comparable efficiency. Thus, PCs are a simpler way to obtain the power of cc(FD).

In the future, we plan to compile Nicolog's WAM instructions directly to C, so its efficiency will become comparable to clp(FD). We also plan to generalize the Nicolog PCs and algorithms further to better exploit 'holes' in domains, so that closer approximations of full arc consistency can be programmed. In particular, we plan to add hierarchical domains [SH92] for large integer domains and real numeric domains. These additions require only small local changes to Nicolog and will not increase the complexity of the whole system by much. In [VHSD93], it is reported that cc(FD) can solve many difficult problems with speed comparable to that of specialized programs painstakingly developed in procedural languages. We expect that with additional reasoning capabilities and better implementation, Nicolog should be able to solve many of these problems in comparable time as well. Real domain variables should make Nicolog better suited for solving engineering problems involving continuous variables, such as engine [Jone90] and automatic transmission [NL93] design.

We also plan to add support for incremental query editing [Have92] for interactive tasks. This means that users, after specifying a query and looking at the solution, will have the option of making arbitrary modifications to the query, looking at the new solutions, and continuing the process indefinitely. After a query edit, the system will not restart the whole computation from scratch, since the system is bound to have done a large amount of work on the user's behalf that is independent of the query edit. Instead, it will try to reuse parts of the proof which are independent of the query changes and make minimal modifications to the parts of the proof which depend on the changes. Implementing this requires maintaining reasons for decisions made by the system and using these reasons to identify parts of proofs which depend on query changes. To implement this, we plan to modify the WAM such that the stack discipline is no longer strictly followed. Incremental query editing will be useful in the implementation of interactive mixed initiative user interfaces for applications such as scheduling and configuration.

References

[AB91] Aggoun, A. and N. Beldiceanu (1991) "Overview of the CHIP Compiler System." In Proc. the Eighth International Conference on Logic Programming, Paris, pp. 775-789.

[AB92] Aggoun, A. and N. Beldiceanu (1992) "Extending CHIP to Solve Complex Scheduling and Packing Problems." In *Journées Francophones de Programmation Logique*, Lille, France.

[AH83] Alefeld, G. and J. Herzberger (1983) *Introduction to Interval Computations*, Computer Science and Applied Mathematics, W. Rheinboldt (ed.), Academic Press, Toronto.

[Aït91] Aït-Kaci, H. (1991) *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Cambridge, MA.

[BB88] Boi, J. M. and F. Benhamou (1988) "Boolean Constraints in Prolog III." Ph.D., Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy.

[BO92] Benhamou, F. and W. J. Older (1992) "Applying Interval Arithmetic to Integer and Boolean Constraints." Technical Report, *Bell Northern Research*, June, 1992.

[Brya86] Bryant, R. E. (1986) "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, **C-35** (3), pp. 677-691.

[Brya92] Bryant, R. E. (1992) "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys*, **24** (3), pp. 293-318.

[BS87] Buttner, W. and H. Simonis (1987) "Embedding Boolean Expressions into Logic Programming," *Journal of Symbolic Computing*, **4** (October), pp. 191-205.

[Buch85] Buchberger, B. (1985) "Grobner Bases: An Algorithmic Method in Polynomial Ideal Theory." In *Multidimentional Systems Theory*, N. K. Bose (ed.), D. Reidel Publishing Co., pp. 184-232.

[Bund84] Bundy, A. (1984) "A Generalized Interval Package and Its Use for Semantic Checking," *ACM Transactions on Mathematical Systems*, **10** (4), pp. 397-409.

[CD93] Codognet, P. and D. Diaz (1993) "Boolean Constraint Solving in clp(FD)." In Proc. *The International Logic Programming Symposium*, MIT Press.

[CFG91] Croft, H., K. Falconer and R. Guy (1991) *Unsolved Problems in Geometry*, Springer Verlag, New York.

[Clea87] Cleary, J. G. (1987) "Logical Arithmetic," *Future Computing Systems*, **2** (2), pp. 125-149.

[Colm90] Colmerauer, A. (1990) "An Introduction to Prolog III," *Communications of the ACM*, **33** (7), pp. 69-90.

[Davi87] Davis, E. (1987) "Constraint Propagation with Interval Labels," *Artificial Intelligence*, **32**, pp. 281-331.

[DC93] Diaz, D. and P. Codognet (1993) "A Minimal Extension of the WAM for clp(FD)." In Proc. *The International Conference on Logic Programming*, MIT Press.

[DEFX92] Durand, J., M. Epstein, E. Freeman and J. Xu (1992) "Integration of Constraint Solvers with the clp(X) Shell: Modularity, Implementation, and Performance Issues." In Proc. 1992 Joint International Conference and Symposium on Logic Programming Post-Conference Workshop on Constraint Logic Programming Systems, Washington DC, pp. 59-75.

[DOW55] Dantzig, G. B., A. Orden and P. Wolfe (1955) "The Generalized Simplex Method for Minimizing Linear Form under Linear Inequality Constraints," *Pacific Journl of Mathematics*, **5** (2), pp. 183-195.

[DP89] Dechter, R. and J. Pearl (1989) "Tree Clustering for Constraint Networks," *Artificial Intelligence*, **38**, pp. 353-366.

[DSVH90] Dincbas, M., H. Simonis and P. V. Hentenryck (1990) "Solving Large Combinatorial Problems in Logic Programming," *Journal of Logic Programming*, **8**, pp. 75-93.

[DVH91] Deville, Y. and P. Van Hentenryck (1991) "An Efficient Arc Consistency Algorithm for a Class of CSP Problems." In Proc. *The International Joint Conference on Artificial Intelligence*, Sydney, Australia, pp. 325-330.

[DVS*88] Dincbas, M., et al. (1988) "The Constraint Logic Programming Language CHIP." In Proc. *The International Conference on Fifth Generation Computer Systems*, Omhsha Publishers, Tokyo, pp. 693-702.

[EK92] Ertl, M. A. and A. Krall (1992) "High-Level Constraints over Finite Domains." Technical Report TR 1851-1992-14, *Institut für Computersprachen, Technische Universität Wien.*

[Freu82] Freuder, E. C. (1982) "A Sufficient Condition for Backtrack-Free Search," *Journal of the ACM*, **29** (1), pp. 24-32.

[Freu85] Freuder, E. C. (1985) "A Sufficient Condition for Backtrack-Bounded Search," *Journal of the ACM*, **32** (4), pp. 755-761.

[GVPZ89] Graf, T., P. Van Hentenryck, C. Pradelles and L. Zimmer (1989) "Simulation of Hybrid Circuits in Constraint Logic Programming." In Proc. *International Joint Conference on Artificial Intelligence*, Detroit.

[Have92] Havens, W. S. (1992) "Intelligent Backtracking in the Echidna Constraint Logic Programming System," *International Journal of Expert Systems*, **5**(4), pp. 319-343.

[HE80] Haralick, R. M. and G. L. Elliot (1980) "Increasing Tree Search Efficiency for Constraint Satisfaction Problems," *Artificial Intelligence*, **8**, pp. 263-313.

[HMS87] Heintze, N. C., S. Michaylov and P. J. Stuckey (1987) "CLP(R) and Some Electrical Engineering Problems." In Proc. *Fourth International Conference on Logic Programming*, MIT Press, Melbourne, pp. 657-703.

[Holz90] Holzbaur, C. (1990) "Specification of Constraint Based Inference Mechanisms through Extended Unification." Ph.D. Thesis., Technischen Universität Wien.

[HSS*92] Havens, W. S., S. Sidebottom, G. Sidebottom, J. Jones and R. Ovans (1992) "Echidna: A Constraint Logic Programming Shell." In Proc. *Pacific Rim International Conference on Aritificial Intelligence*, Seoul, Korea, pp. 165-171.

[JL87] Jaffar, J. and J.-L. Lassez (1987) "Constraint Logic Programming." In Proc. *Fourteenth ACM Symposium on the Principles of Programming Languages*, Munich, pp. 111-119.

[JM87] Jaffar, J. and S. Michaylov (1987) "Methodology and Implementation of a CLP System." In Proc. *Fourth International Conference on Logic Programming*, Melbourne, Australia, pp. 111-119.

[JMSY92] Jaffar, J., S. Michaylov, P. J. Stuckey and R. H. C. Yap (1992) "An Abstract Machine for CLP(*R*)." In Proc. *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, ACM Press, San Francisco, pp. 128-139.

[Jose92] Joseph, S. (1992) "IntelCAD: Intelligent Computer-Aided Design," *The BC Professional Engineer*, **43** (5), pp. 8-11.

[Jone90] Jones, J. D. (1990) "Optimisation of Stirling Engine Regenerator Design." In Proc. *1990 IECEC, American Society of Chemical Engineers*, Reno, NV, pp. 359-365.

[Laur78] Lauriere, J.-L. (1978) "A Language and a Program for Stating and Solving Combinatorial Problems," *Artificial Intelligence*, **10**, pp. 29-127.

[Lloy84] Lloyd, J. W. (1984) *Foundations of Logic Programming*, Symbolic Computation, D. W. Loveland (ed.), Springer-Verlag, New York.

[LMY87] Lassez, C., K. McAloon and R. Yap (1987) "Constraint Logic Programming in Options Trading," *IEEE Expert* **4** (2), pp. 11-18.

[Mack77] Mackworth, A. K. (1977) "Consistency in Networks of Relations," *Artificial Intelligence*, **8**, pp. 99-118.

[MF85] Mackworth, A. K. and E. C. Freuder (1985) "The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems," *Artificial Intelligence*, **25**, pp. 65-74.

[MH86] Mohr, R. and T. C. Henderson (1986) "Arc and Path Consistency Revisited," *Artificial Intelligence*, **28**, pp. 225-233.

[Nade89] Nadel, B. A. (1989) "Constraint Satisfaction Algorithms," *Computational Intelligence*, **5**, pp. 188-224.

[NL93] Nadel, B. A. and J. Lin (1993) "Automobile Transmission Design: Constraint Satisfaction Formulation and Prolog Implementation," *Expert Systems: Research and Applications*, to appear.

[OV90] Older, W. and A. Vellino (1990) "Extending Prolog with Constraint Arithmetic on Real Intervals." In Proc. *The Canadian Conference on Computer and Electrical Engineering*, Ottawa.

[SA89] Sakai, K. and A. Aiba (1989) "CAL: A Theoretical Background of Constraint Logic Programming and its Applications," *Journal of Symbolic Computation*, **8** (6), pp. 589-603.

[SH92] Sidebottom, G. and W. S. Havens (1992) "Hierarchical Arc Consistency for Disjoint Real Intervals in Constraint Logic Programming," *Computational Intelligence*, **8** (4), pp. 601-623.

[Side93] Sidebottom, G. (1993) "Implementing CLP(B) with the Connection Theorem Proving Method and a Reason Maintenance System," *Journal of Symbolic Computation*, **15**, pp. 27-48.

[Simo89] Simonis, H. (1989) "Test Generation Using the Constraint Logic Programming Language CHIP." In Proc. *The Sixth International Conference on Logic Programming*, Lisbon, Portugal, pp. 101-112.

[SS86] Sterling, L. and E. Shapiro (1986) *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, MA.

[VanH89] Van Hentenryck, P. (1989) *Constraint Satisfaction in Logic Programming*, The MIT Press, Cambridge.

[VHD91] Van Hentenryck, P. and Y. Deville (1991) "The Cardinality Operator: A New Logical Connective for Constraint Logic Programming." In Proc. *The International Conference on Logic Programming*, MIT Press, Paris, France.

[VHSD91] Van Hentenryck, P., V. Saraswat and Y. Deville (1991) "Constraint Processing in cc(FD)." Technical Report, *Brown University.*

[VHSD93] Van Hentenryck, P., V. Saraswat and Y. Deville (1993) "Design, Implementation, and Evaluation of the Constraint Language cc(FD)." Technical Report, *Brown University.*

[Wali89] Walinsky, C. (1989) "Constraint Logic Programming with Regular Sets." In Proc. *Sixth International Conference on Logic Programming*, Lisbon, Portugal, pp. 181-196.

[Warr83] Warren, D. H. D. (1983) "An Abstract Prolog Instruction Set." Technical Note 309, *SRI International.*

[Wolf91] Wolfram, S. (1991) *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Don Mills, ON.

A. Concise Overview of Nicolog

This appendix gives a concise overview of how Nicolog extends Prolog with constraints. The syntax of Nicolog is the same as the syntax of the Edinburgh family of Prologs as described in [SS86]. Nicolog uses some infix and prefix operators for constraints. The operator syntax can be described with the `op/3` predicate which is standard in Edinburgh Prolog. Table 1 gives the operator precedences for the constraint symbols used in Nicolog. They are defined with the standard Prolog predicate

```
op(Precedence, Associativity, Symbols).
```

Lower precedence numbers mean stronger binding power. Thus `A+B*C` means `A+(B*C)`. The associativity symbols `fx` and `fy` are for unary prefix operators while `xfx`, `xfy`, and `yfx` are for binary infix operators which are non, right, and left associative, respectively. Thus, `X-Y-Z` means `(X-Y)-Z` and `A,B,C` means `A,(B,C)`. `A#=B#=C` is a syntax error. See [SS86] for a more complete description.

```
op(1100, xfy, [;])
op(1050, xfy, [->])
op(1000, xfy, [,])
op( 900,  fy, [\,\\])
op( 700, xfx, [=:=,=\=,<,=<,>,>=])
op( 700, xfx, [#=,#/=,#<,#>,#=<,#>=])
op( 700, xfx, [$=,$/=,$<,$>,$=<,$>=])
op( 550, xfy, [:])
op( 525, xfx, [..])
op( 520, yfx, [<=>])
op( 510, yfx, [=>])
op( 505,  fx, [<<,>>])
op( 500, yfx, [+,-,\/,#])
op( 500,  fx, [-])
op( 500,  fx, [min,max])
op( 400, yfx, [*,/,/\])
op( 300, xfx, [mod])
op( 200, xfy, [^])
op( 100,  fx, [<,>])
```

Table 1. Precedence of Nicolog constraint symbols

The following tables give the extended BNF syntax for the domain, primitive, and projection constraints in Nicolog. Each syntax table gives a brief description of the meaning of each construct. For more details, see section 2.1.

Table 2 gives the syntax of domain constraints. Table 2 only gives some of the rules for *set* and *expr*. The other rules for these categories define constructs which are normally only used in projection constraints. The complete sets of rules for the *set* and *expr* categories are given in table 4. The syntax for primitive constraints is given in table 3 and the syntax for PCs is given in table 4.

dconstr ::=	*term* : *set*	(domain declaration)
term ::=	*any Edinburgh syntax prolog term*	
set ::=	*range*	(single range, eg. `1..5`)
	variable	(domain of variable)
	{*range*, ...}	(union of ranges)
	\ *set*	(complement)
	...	
range ::=	*expr* . . *expr*	(multiple element range)
	expr	(≡ *expr* . . *expr*)
expr ::=		(expression)
	variable	(instantiated variable)
	number	(constant)
	`inf`	(infinity)
	- *expr*	(negate)
	expr + *expr*	(add)
	expr - *expr*	(subtract)
	expr * *expr*	(multiply)
	expr / *expr*	(divide)
	expr ^ *expr*	(exponentiation)
	`root` (*expr*, *expr*)	(root extraction)
	`log` (*expr*, *expr*)	(logarithm)
	`min` [*expr*, ...]	(minimum of sequence)
	`max` [*expr*, ...]	(maximum of sequence)
	...	

Table 2. Syntax of Nicolog domain constraints

pconstr ::=		(primitive constraint)
	cterm #= *cterm*	(equal)
	cterm #/= *cterm*	(not equal)
	cterm #=< *cterm*	(less or equal)
	cterm #< *cterm*	(less)
	cterm #>= *cterm*	(greater or equal)
	cterm #> *cterm*	(greater)
	cterm /\ *cterm*	(and)
	cterm \/ *cterm*	(inclusive or)
	~ *cterm*	(not)
	cterm => *cterm*	(implies)
	cterm <=> *cterm*	(equivalent)
	cterm # *cterm*	(exclusive or)
cterm ::=		(constraint term–argument to constraint)
	variable	
	number	
	- *cterm*	(negate)
	cterm + *cterm*	(add)
	cterm - *cterm*	(subtract)
	cterm * *cterm*	(multiply)
	cterm / *cterm*	(divide)
	cterm ^ *cterm*	(exponentiation)
	`root(`*cterm*`,`*cterm*`)`	(root extraction)
	`log(`*cterm*`,`*cterm*`)`	(logarithm)
	`abs(`*cterm*`)`	(absolute value)
	`min(`*cterm*`,`*cterm*`)`	(minimum)
	`max(`*cterm*`,`*cterm*`)`	(maximum)
	`cond(`*cterm*`,`*cterm*`,`*cterm*`)`	(conditional expression)
	pconstr	(nested constraint)

Table 3. Syntax of Nicolog primitive constraints

proj ::=		(projection constraint)
	variable $= *set*	(means *variable* ∈ *set*)
	variable $=< *expr*	(≡ *variable* $= -inf..*expr*)
	variable $>= *expr*	(≡ *variable* $= *expr*..inf)
set ::=	*range*	(single range)
	variable	(≡ (<*variable*)..(>*variable*))
	{}	(fail set)
	{*range*,...}	(union of ranges)
	\ *set*	(complement)
	\\ *expr*	(≡ \ *expr*..*expr*)
	expr -> *set* ; *set*	(conditional set)
	b(*set*, *set*, *set*, *set*)	(Boolean conditional set)
	b1(*set*, set_1, set_2)	(≡ b(*set*, set_1, set_2, set_2))
	b2(*set*, set_2, set_1)	(≡ b(*set*, set_1, set_2, set_1))
range ::=	*expr* .. *expr*	(multiple element range)
	expr	(≡ *expr*..*expr*)
expr ::=		(expression)
	variable	(instantiated variable)
	number	(constant)
	inf	(infinity)
	<*set*	(lower bound)
	>*set*	(upper bound)
	fail	(fail expression)
	<< *expr*	(just smaller)
	>> *expr*	(just bigger)
	- *expr*	(negate)
	expr + *expr*	(add)
	expr - *expr*	(subtract)
	expr * *expr*	(multiply)
	expr / *expr*	(divide)
	expr ^ *expr*	(exponentiation)
	root(*expr*, *expr*)	(root extraction)
	log(*expr*, *expr*)	(logarithm)
	min [*expr*,...]	(minimum of sequence)
	max [*expr*,...]	(maximum of sequence)
	expr , *expr*	(and test)
	expr ; *expr*	(or test)
	expr =:= *expr*	(equal test)
	expr =\= *expr*	(not equal test)
	expr =< *expr*	(less or equal test)
	expr < *expr*	(less test)
	expr >= *expr*	(greater or equal test)
	expr > *expr*	(greater test)
	expr -> *expr* ; *expr*	(conditional expression)
	b(*set*, *expr*, *expr*, *expr*)	(Boolean conditional expression)
	b1(*set*, $expr_1$, $expr_2$)	(≡ b(*set*, $expr_1$, $expr_2$, $expr_2$))
	b2(*set*, $expr_2$, $expr_1$)	(≡ b(*set*, $expr_1$, $expr_2$, $expr_1$))

Table 4. Syntax of Nicolog projection constraints (PCs)

B. Compiling Multiplication and Division

The following are the full definitions used to compile multiplication and division into PCs.

```
pr (A*B) =
      LBA = (<pr A),
      UBA = (>pr A),
      LBB = (<pr B),
      UBB = (>pr B),
      AZero = (LBA =:= 0, UBA =:= 0),
      ANotPos = (UBA =< 0),
      ANotNeg = (LBA >= 0),
      BZero = (LBB =:= 0, UBB =:= 0),
      BNotPos = (UBB =< 0),
      BNotNeg = (LBB >= 0),
      Bs = [LBA*LBB,LBA*UBB,UBA*LBB,UBA*UBB],
        AZero -> 0
      ; BZero -> 0
      ; ANotPos, BNotPos -> UBA*UBB .. LBA*LBB
      ; ANotPos, BNotNeg -> UBA*LBB .. LBA*UBB
      ; ANotNeg, BNotPos -> LBA*UBB .. UBA*LBB
      ; ANotNeg, BNotNeg -> LBA*LBB .. UBA*UBB
      ; min Bs..max Bs

pr (A/B) =
      LBA = (<pr A),
      UBA = (>pr A),
      LBB = (<pr B),
      UBB = (>pr B),
      AZero = (LBA =:= 0, UBA =:= 0),
      ANeg = (UBA < 0),
      APos = (LBA > 0),
      BZero = (LBB =:= 0, UBB =:= 0),
      BNeg = (UBB < 0),
      BPos = (LBB > 0),
      Bs = [LBA/LBB,LBA/UBB,UBA/LBB,UBA/UBB],
        AZero -> 0
      ; BZero -> {}   % division by 0 fails
      ; ANeg, BNeg -> UBA/LBB .. LBA/UBB
      ; ANeg, BPos -> UBA/UBB .. LBA/LBB
      ; APos, BNeg -> LBA/LBB .. UBA/UBB
      ; APos, BPos -> LBA/UBB .. UBA/LBB
      ; ( BZero -> {} % division by 0 fails
        ; LBB=<0 , UBB>=0 -> -inf..inf
        ; min Bs..max Bs)
```

The start with the definition of common subexpressions and then proceed with a case analysis of the arguments. Nicolog does not use these definitions directly at runtime. Instead, it infers maximal domains from information which is available at compile time, such as domain constraints and variables occuring in calls to built in predicates such as `is/2`. Then, for a given multiplication

or division, the case that applies at compile time is used in the compilation. For instance, the constraint in

```
p(A,B,C) :-
    A:1..10,
    B:0..10,
    C:1..10,
    A*B #= C.
```

has the following isolations:

```
C #= A*B,
A #= C/B, and
B #= C/A
```

To project onto C, we need to evaluate *pr* (A*B). Since we know from the domain constraints that both A and B are non-negative, this compiles into (<A) * (<B) .. (>A) * (>B). To project onto A, we need to evaluate *pr* (C/B). Since the domain constraints allow B to possibly be zero, the final general case is the result[20]:

```
( (<B) =:= 0 , (>B) =:= 0) -> ()
; (<B) =< 0, (>B) >= 0 -> -inf..inf
; min[...] .. max[...] )
```

Since A and C are both know to be positive, projecting onto B gives (<C) / (>A) .. (>C) / (<A).

[20]It is actually possible to give even more specialized rules for division which consider when one argument contains zero and the other does not. It is even possible to perform arc consistency on constraints at compile time to infer smaller maximal domains for the variables. In the example here, this would lead to the fact that B cannot be zero. The Nicolog compiler currently only does the analysis described in the main text here.

C. Schur Lemma Program

This appendix contains the Nicolog program used to generate the results given in section 5.3. It was translated from a clp(FD) program which was kindly supplied by Daniel Diaz. Recall that the problem is to try to put n balls labelled by the integers $\{1,\ldots,n\}$ into three boxes so that for any triple (x,y,z) such that $x + y = z$, balls x, y, and z are not all in the same box. The following program formulates the problem as a matrix M_{ij} $(1 \le i \le n,\ 1 \le j \le 3)$ (implemented with a list of lists) where M_{ij} is true iff ball i is in box j. The `only1/1` predicate sets up the constraints that force each ball to be in exactly one box:

$$M_{i1} \ + \ M_{i2} + \ M_{i3} \ \#= \ 1 \ (1 \le i \le n).$$

The other constraints, that for each (x,y,z) such that $x + y = z$

$$\sim (M_{xj} \ /\backslash \ M_{yj} \ /\backslash \ M_{zj} \ ) \ (1 \le j \le 3)$$

for the requirement that these balls are not all in the same box, are enforced by the `constraints/2` predicate. This predicate optimizes the case where $x = y$ by simply using

$$\sim (M_{xj} \ /\backslash \ M_{zj} \ ) \ (1 \le j \le 3).$$

```
schur(N,A):-
    create_array(N,3,A),
    A:0..1,
    only1(A),
    constraints(A,A),
    array_labeling(A).

only1([]).
only1([[I1,I2,I3]|A]) :-
    I1 + I2 + I3 #= 1,
    only1(A).

constraints([],_).
constraints([_],_).
constraints([_,[K1,K2,K3]|A2],[[I1,I2,I3]|A1]):-
    ~(I1 /\ K1),
    ~(I2 /\ K2),
    ~(I3 /\ K3),
    triplet_constraints(A2,A1,[I1,I2,I3]),
    constraints(A2,A1).

triplet_constraints([],_,_).
triplet_constraints([[K1,K2,K3]|A2],
                    [[J1,J2,J3]|A1],
                    [I1,I2,I3]):-
    ~(I1 /\ J1 /\ K1),
    ~(I2 /\ J2 /\ K2),
```

```
    ~(I3 /\ J3 /\ K3),
    triplet_constraints(A2,A1,[I1,I2,I3]).

create_array(NR,NC,A):-
    length(A,NR),
    create_array1(A,NC).

create_array1([],_NC).
create_array1([R|Rs],NC) :-
    length(R,NC),
    create_array1(Rs,NC).

array_labeling([]).
array_labeling([L|A]):-
    label(L),
    array_labeling(A).

label([]).
label([X|L]) :-
    indomain(X),
    label(L).
```

D. Bridge Construction Scheduling Program

This appendix contains the Nicolog program used to generate the results in 5.6. It is a solution to the bridge scheduling problem given in [VanH89] and was translated from a CLP(BNR) program which was kindly provided by Frédéric Benhamou.

```
% the following four predicates define the problem

% define(K,Xstop) K is a list of tasks where each task
%    consists of a name, variable start time, and fixed
%    duration Xstop is the start time of the psueudo task
%    which signals project completion
define(K,Xstop):-
    K = [[start,X0,0],[a1,Xa1,4],[a2,Xa2,2],[a3,Xa3,2],
         [a4,Xa4,2],[a5,Xa5,2],[a6,Xa6,5],[p1,Xp1,20],
         [p2,Xp2,13],[ue,Xue,10],[s1,Xs1,8],[s2,Xs2,4],
         [s3,Xs3,4],[s4,Xs4,4],[s5,Xs5,4],[s6,Xs6,10],
         [b1,Xb1,1],[b2,Xb2,1],[b3,Xb3,1],[b4,Xb4,1],
         [b5,Xb5,1],[b6,Xb6,1],[ab1,Xab1,1],[ab2,Xab2,1],
         [ab3,Xab3,1],[ab4,Xab4,1],[ab5,Xab5,1],[ab6,Xab6,1],
         [m1,Xm1,16],[m2,Xm2,8],[m3,Xm3,8],[m4,Xm4,8],
         [m5,Xm5,8],[m6,Xm6,20],[l1,Xl1,2],[t1,Xt1,12],
         [t2,Xt2,12],[t3,Xt3,12],[t4,Xt4,12],[t5,Xt5,12],
         [ua,Xua,10],[v1,Xv1,15],[v2,Xv2,10],[k1,Xk1,0],
         [k2,Xk2,0],[stop,Xstop,0]],
    L =  [X0,Xa1,Xa2,Xa3,Xa4,Xa5,Xa6,Xp1,Xp2,Xue,Xs1,Xs2,Xs3,
          Xs4,Xs5,Xs6,Xb1,Xb2,Xb3,Xb4,Xb5,Xb6,Xab1,Xab2,Xab3,
          Xab4,Xab5,Xab6,Xm1,Xm2,Xm3,Xm4,Xm5,Xm6,Xl1,Xt1,Xt2,
          Xt3,Xt4,Xt5,Xua,Xv1,Xv2,Xk1,Xk2,Xstop],
    L : 0..120.

% liste_prec(L) L is a list of pairs of tasks where the first
%    task precedes the second
liste_prec(L):-
    L = [[start,a1],[start,a2],[start,a3],[start,a4],
         [start,a5],[start,a6],[start,ue],[a1,s1],
         [a2,s2],[a5,s5],[a6,s6],[a3,p1],[a4,p2],
         [p1,s3],[p2,s4],[p1,k1],[p2,k1],[s1,b1],
         [s2,b2],[s3,b3],[s4,b4],[s5,b5],[s6,b6],
         [b1,ab1],[b2,ab2],[b3,ab3],[b4,ab4],[b5,ab5],
         [b6,ab6],[ab1,m1],[ab2,m2],[ab3,m3],[ab4,m4],
         [ab5,m5],[ab6,m6],[m1,t1],[m2,t1],[m2,t2],
         [m3,t2],[m3,t3],[m4,t3],[m4,t4],[m5,t4],
         [m5,t5],[m6,t5],[m1,k2],[m2,k2],[m3,k2],
         [m4,k2],[m5,k2],[m6,k2],[l1,t1],[l1,t2],
         [l1,t3],[l1,t4],[l1,t5],[t1,v1],[t5,v2],
         [t2,stop],[t3,stop],[t4,stop],[v1,stop],
         [v2,stop],[ua,stop],[k1,stop],[k2,stop]].

% list_disjunction(L) L is a list of lists of disjuctive
%    task names.  The tasks in a sublist must not overlap in
%    time.
list_disjunction(L):-
    L = [[v1,v2],[l1,t1,t2,t3,t4,t5],[m1,m2,m3,m4,m5,m6],
```

```
        [s1,s2,s3,s4,s5,s6],[p1,p2],[a1,a2,a3,a4,a5,a6],
        [b1,b2,b3,b4,b5,b6]].

% addcons_list(L) L is a list of pairs of tasks which satisfy
%    distance constraints.  For instance, [ee,s1,b1,4] means
%    the end time of s1 must be 4 units of time before the
%    end time of b1.
addcons_list(L):-
    L = [[ee,s1,b1,4],[ee,s2,b2,4],[ee,s3,b3,4],
         [ee,s4,b4,4],[ee,s5,b5,4],[ee,s6,b6,4],
         [ss,s1,ue,6],[ss,s2,ue,6],[ss,s3,ue,6],
         [ss,s4,ue,6],[ss,s5,ue,6],[ss,s6,ue,6],
         [se,s1,a1,3],[se,s2,a2,3],[se,s3,p1,3],
         [se,s4,p2,3],[se,s5,a5,3],[se,s6,a6,3],
         [es,ua,m1,-2],[es,ua,m2,-2],[es,ua,m3,-2],
         [es,ua,m4,-2],[es,ua,m5,-2],[es,ua,m6,-2]].

% exact_day_list(L) L is a list of tasks which have to start
%    on a specific day.
exact_day_list([[11,30]]).

% this is the top level predicate for the program
bridge :-
    T is cputime,
    define(K,Xstop),
    constraints(K,Bs),
    T2 is cputime - T,
    write('setup '), write(T2), write(ms), nl,
    minimize(Bs,Xstop,MinCompletion),
    T1 is cputime - T,
    write('total '), write(T1), write(ms), nl.

constraints(K,B):-
    precedence(K),
    extra_constraints(K),
    disjunctions(K,B).

% Precedence Constraints
precedence(K):-
    liste_prec(L),
    precede(K,L).

precede(K,[]).
precede(K,[[T1,T2]|Ts]):-
    find(K,[T1,X1,D1]),
    find(K,[T2,X2,D2]),
    X1+D1 #=< X2,
    precede(K,Ts).

% Extra Constraints
extra_constraints(K):-
    addcons_list(L),
    addcons(K,L),
    exact_day_list(L1),
    exact_day(K,L1).
```

```
% Distance constraints
addcons(K,[]).
addcons(K,[[C,T1,T2,N]|Ls]):-
    constype(K,C,T1,T2,N),
    addcons(K,Ls).

constype(K,ee,T1,T2,N):-
    find(K,[T1,X1,D1]),
    find(K,[T2,X2,D2]),
    X1+D1 #=< X2+D2+N
constype(K,ss,T1,T2,N):-
    find(K,[T1,X1,D1]),
    find(K,[T2,X2,D2]),
    X2+N #=< X1
constype(K,se,T1,T2,N):-
    find(K,[T1,X1,D1]),
    find(K,[T2,X2,D2]),
    X1 #=< X2+D2+N
constype(K,es,T1,T2,N):-
    find(K,[T1,X1,D1]),
    find(K,[T2,X2,D2]),
    X1 #>= X2+D2+N

% Exact Day Constraints
exact_day(K,[]).
exact_day(K,[[T,N]|L]):-
    find(K,[T,X,D]),
    X #= N,
    exact_day(K,L).

% Disjunctive Constraints
disjunctions(K,Bs):-
    list_disjunction(D),
    disj_constraints(K,D,Bs,[]).

disj_constraints(K,[],Bs,Bs).
disj_constraints(K,[D|Ds],Bs1,Bs):-
    disjunction(K,D,Bs1,Bs2),
    disj_constraints(K,Ds,Bs2,Bs).

disjunction(L,[],Bs,Bs).
disjunction(L,[T1|Ts],Bs1,Bs):-
    find(L,[T1,X1,D1]),
    disj(L,X1,D1,Ts,Bs1,Bs2),
    disjunction(L,Ts,Bs2,Bs).

disj(L,X1,D1,[],Bs,Bs).
disj(L,X1,D1,[T2|Ts],[B|Bs1],Bs):-
    find(L,[T2,X2,D2]),
    B #= (X1 #>= X2+D2),
    B + (X2-D1 #>= X1) #= 1,
    disj(L,X1,D1,Ts,Bs1,Bs).

% minimize(Vars,X,XMin) XMin is the smallest value for X over
%   all instantiation of variables in Vars
```

```
minimize(Vars,X,XMin):-
    retractall(upperBound(_)),
    assert(upperBound(9999999)),
    repeat,
    upperBound(XMin),
    (
      X #< XUB,
      labelff(Vars) ->
        X:XLB..._,
        write('Best solution so far, Min = '), write(X), nl
        retract(upperBound(_)),
        assert(upperBound(XLB)),
        fail
    ;
        !
    ).

% label variables using first fail principle--try variables
% with smallest domains first
labelff([]).
labelff([X|L]) :-
    deleteff([X|L],V,R),
    indomain(V),
    labelff(R).

% Utilities

find([[T1,X1,D1]|Ts],[T1,X1,D1]):-!.
find([T|Ts],T1):- find(Ts,T1).

repeat.
repeat:-repeat.
```