Logic, Programming and Prolog (Supplement)

Ulf Nilsson
Dept of Computer and Information Science
Linköping University

ulfni@ida.liu.se
http://www.ida.liu.se/labs/logpro/ulfni

Contents

Chapter 16

Finite Domain Constraints

In this chapter we study a specific constraint logic programming language with many industrial applications – in particular combinatorial optimization problems such as scheduling, placement, planning and resource allocation. The constraints are called *finite domain* constraints and are arithmetic relations (equalities, inequalities and disequalities) involving integer variables ranging over finite sets of integers. However, CLP(FD) systems often contain also more powerful relations – sometimes called *global constraints* – that are extremely expressive and are supported by specialized solvers.

16.1 Primitive Finite Domain Constraints

In what follows a *finite domain* is a finite subset of the set of all integers, and a *finite domain variable* is an integer variable ranging over a finite domain. The domain of a finite domain variable (or simply domain variable) can be specified explicitly; for instance as in:

```
| ?- X in 3..8.
```

Here we say that the value of `X` is a member in the finite set $\{3, \ldots, 8\}$. The expression `X in 3..8` is our first example of a *finite domain constraint* – it constrains the domain of `X` to a finite subset of the integers. Note that a domain does not have to be an interval; it is possible to say for instance:

```
| ?- X in {1,3,5}.
```

It is also possible to specify the domain of a domain variable implicitly using arithmetic relations involving finite domains, other domain variables and operations on integers. Consider the following query:

```
| ?- X in 3..8, Y in 1..4, Z #= X+Y.
```

Here we define the domain variables `X` and `Y` explicitly but the domain of the domain variable `Z` is given implicitly by the finite domain constraint `Z #= X+Y` meaning that `Z` is the sum of `X` and `Y`. Since `X` is at most 8 and `Y` is at most 4, `Z` is at most 12. Similarly, the least value taken by `Z` is 4 (i.e. 3+1). Hence, the domains of the variables are:

```
X in 3..8,
Y in 1..4,
Z in 4..12
```

This is also the result printed by SICStus Prolog in response to the query above. Most of the standard arithmetic operations and relations can be used in finite domain constraints; for instance, addition (`+`), subtraction (`-`), multiplication (`*`), integer division (`/`), equality (`#=`), disequality (`#\=`) and the standard inequalities (`#<`, `#>`, `#=<` and `#>=`).

One should not confuse the *domains* of domain variables with *solutions* of the constraints. The domain of a variable is the set of all *possible* values of the variable, but not all possible values are solutions – in the previous example 4 is in the domain of both `X`, `Y` and `Z`, but it is no *solution* to the constraints. By a solution we mean an assignment of an integer to each variable occurring in the constraints. That is, a function from variables to integers, or – as we sometimes call it – a *valuation*.

Note that if we impose additional constraints, the domain of a domain variable may shrink. Consider the query:

```
| ?- X in 0..9, Y in 5..7, X #> Y.
```

Here the domains of `X` and `Y` are:

```
X in 6..9,
Y in 5..7
```

Initially the domain of `X` is $\{0, \ldots, 9\}$ but the constraint `X #> Y` restricts the domain of `X` to $\{6, \ldots, 9\}$. In our case we see that:

- the constraint `X in 0..9` has finitely many solutions in `X`;
- the constraint `Y in 5..7` also has finitely many solutions, but in `Y`;
- the constraint `X #> Y` has infinitely many solutions one of which is $\{\texttt{X} \mapsto 6, \texttt{Y} \mapsto 5\}$.

Together the constraints have only a finite number of solutions – for instance $\{\texttt{X} \mapsto 6, \texttt{Y} \mapsto 5\}$ and $\{\texttt{X} \mapsto 9, \texttt{Y} \mapsto 7\}$.

Even if the constraints above do have solutions the constraints are not sufficiently specific to infer unique solutions for `X` and `Y`. A domain variable remains unknown until its domain is a singleton as in:

```
| ?- X in 0..9, Y in 0..1, X #< Y.
```

Since `Y` is at most 1 we can infer that `X` must be 0 and that the domain of `Y` is 1. In such a case we say `X` and `Y` are *ground* or *determined*.

It may also happen that the domain of a variable is the empty set. For instance, consider:

```
| ?- X in 4..6, Y in 1..3, X #< Y.
```

Here `X` cannot be less than 4 and `Y` cannot be greater than 3, hence `X` cannot be less than `Y`. A combination of constraints where some variable has the empty domain is said to be *unsatisfiable*.

Finite domain constraint systems are usually *incomplete* – that is, the solver is in general unable to determine if a constraints store is satisfiable or not. In principle it is not difficult to check this – since every variable has a finite domain it is in principle possible to check all possibilities. However, this is usually not a feasible strategy since it leads to an expontial algorithm. Instead most finite domain solvers check satisfiability by checking the domains of variables – if some variable has an empty domain the constraint store is bound to be unsatisfiable. However, if all variables have a non-empty domain the solver in general does not know.

Finite domain systems are also incomplete in the sense that they do not always deduce the smallest possible domain of the variables. This is illustrated by the following constraints:

```
| ?- X in 1..12, Y in 1..12, X #= 2*Y.
```

There are only six solutions to the equations – the domain of `Y` is $\{1, \ldots, 6\}$ and the domain of `X` is $\{2, 4, 6, 8, 10, 12\}$. However, the output produced by e.g. SICStus Prolog is in fact:

```
X in 2..12,
Y in 1..6
```

The next example illustrates constraints that have no solutions but where SICStus Prolog is unable to detect this due to the incomplete solver:

```
| ?- X in 1..2, Y in 1..2, Z in 1..2, X #\= Y, X #\= Z, Y #\= Z.
```

```
X in 1..2,
Y in 1..2,
Z in 1..2
```

To get around the incompleteness finite domain systems typically contain mechanisms to explicitly enumerate the solutions to a set of constraints. For instance, SICStus Prolog contains a built-in predicate `labeling`/2 for enumerating solutions. The predicate takes two arguments:

- the first is a list of search options that determine the order in which the solutions are enumerated (and if a solution should be enumerated at all);
- the second is a list of domain variables for which solutions are sought after.

For instance, if we want to enumerate the solutions to the previous constraints (using the default search options) we may write as follows in SICStus Prolog:

```
| ?- X in 1..12, Y in 1..12, X #= 2*Y, labeling([],[X,Y]).
```

The result is the six (expected) solutions.

Finite domain systems usually have constraints to specify the domain of several domain variables simultaneously. For instance, in SICStus Prolog the domains of several domain variables can be specified simultaneously using the constraint `domain`/3. In the following query we set the domain of the variables `X`, `Y`, `Z` to the domain $\{0, \ldots, 9\}$:

```
| ?- domain([X,Y,Z], 0, 9).
```

16.2 CLP(FD)

By combining finite domain constraints with logic programming we obtain a *programmable constraint solver* – the constraints define a solution space and logic programming can be used to search non-deterministically in the solution space. In this chapter we illustrate some benefits of combining the two. We also discuss and analyze methodologies for writing CLP(FD) programs.

Consider first the problem of positioning N queens on an $N \times N$ chessboard in such a way that no queen is attacked by any other queen. Since there are N queens and N columns it follows that there must be exactly one queen in each column. Similarly there must be exactly one queen in each row. Hence

any solution to the problem can be encoded as a list $[R_1, \ldots, R_N]$ where R_i is the row position of the queen in column i. It also follows that $R_1, \ldots, R_N$ must be distinct integers in the range $1, \ldots, N$. (That is, $[R_1, \ldots, R_N]$ must be a permutation of the list $[1, \ldots, N]$.) The problem can be described as follows in the language CLP(FD):

```
1. queens(N, L) :-
2.     length(L, N),
3.     domain(L, 1, N),
4.     safe(L),
5.     labeling([], L).
```

The predicate `queen(N, L)` states that `L` (a list of integers) is a solution to the N-queens problem. A solution is established by saying that `L` is a list of length `N` (line 2) where each element is a distinct domain variable with domain $\{1, \ldots, N\}$ (line 3). The predicate `length(L,N)` is a built-in predicate that relates a list `L` with the length `N` of the list. If `length`/2 is called with a variable and a natural number N, the variable will be bound to the most general list of length N upon success (i.e. a list consisting of distinct variables). Line 4 is used to constrain the domain variables in such a way that no queen may attack any other queen. The predicate `safe`/1 is defined as follows:

```
 6. safe([]).
 7. safe([X|Xs]) :-
 8.     safe_between(X, Xs, 1),
 9.     safe(Xs).

10. safe_between(X, [], M).
11. safe_between(X, [Y|Ys], M) :-
12.     no_attack(X, Y, M),
13.     M1 is M+1,
14.     safe_between(X, Ys, M1).

15. no_attack(X, Y, N) :-
16.     X #\= Y, X+N #\= Y, X-N #\= Y.
```

Consider first the clause in line 15-16: Two queens (on rows `X` and `Y`) that are `N` columns apart do not attack each other if they are neither on the same row (`X #\= Y`) nor on the same diagonal (`X+N #\= Y` and `X-N #\= Y`).

To simplify the presentation we use the term *suffix* to denote the right-most part of a board – i.e. if we have a board sized $N \times N$ a suffix is either the

empty board or column N or columns $N-1$ to N and so on. Now consider a situation where we have a suffix `Xs` of the board and a queen in row `X`, `N` columns to the left of the suffix. Now the predicate `safe_between(X,Xs,M)` is meant to hold when the queen in row `X` does not attack any of the queens in the suffix `Xs`. If the suffix is empty this is trivially satisfied (line 10). But if the suffix is non-empty we require that the queen in row `X` does not attack the leftmost queen in the suffix (`M` columns to the right), (line 12), and that the queen in row `X` does not attack any queen in the smaller prefix which is `M+1` columns to the right (lines 13-14).

Now the predicate `safe`/1 is supposed to hold if no queen attacks any other queen. If the board is empty the board is trivially safe (line 6). In case of a non-empty board `[X|Xs]` the board is safe if the leftmost queen does not attack any queen in the suffix that starts immediately in the next column (line 8) and the board `Xs` is safe (line 9).

The main predicate of the previous example illustrates a prototypical methodology for describing problems in CLP(FD). Schematically CLP(FD) programs typically look as follows:

```
1. solution(L) :-
2.     create_variables(L),
3.     constrain_variables(L),
4.     solve_constraints(L).
```

The first part of the program is responsible for creating domain variables and their domains. The second step is to constrain the domain variables (line 3), and finally solutions to the constraints are enumerated (line 4).

16.3 Optimization

It is not always desirable to report all solutions to a set of constraints. There are problems where we are rather interested in reporting optimal (minimal or maximal) solutions. Optimality is generally expressed in terms of some cost function – or more precisely, in terms of a domain variable that equates some cost function. In SICStus Prolog there are two ways of optimizing such a *cost variable*; both of which are closely related to enumeration of solutions to a constraint store. The first option is used when seeking an optimal value of a cost variable *for a given constraint store*. For such a "local" optimization problem we may simply use the predicate `labeling`/2:[1]

```
| ?- Goal, labeling([maximize(X)],Vars).
```

[1]There is a similar option for minimizing the value of a cost variable.

Beware that a goal such as that above may produce more than one maximal value for X – if *Goal* is non-deterministic there will be one maximal value for each constraint store induced by *Goal*. Consider the program:

```
p(X,Y) :- X #< Y.
p(X,Y) :- X #< Y+1.
```

The following goal yields two optimal answers:

```
| ?- X in 1..10, Y in 2..5, p(X,Y), labeling([maximize(X)],[X]).

X = 4, Y = 5
X = 5, Y = 5
```

The second option is used when seeking the optimal value of a cost variable X subject to *any* constraint store induced by a given goal *Goal*:

```
| ?- maximize(Goal,X).
| ?- minimize(Goal,X).
```

In SICStus it is required that each refutation of *Goal* assigns a unique integer to X (for instance, using the predicate `labeling`/2).

A predicate such as `maximize`/2 is typically implemented by repeatedly calling *Goal*, each time with an additional constraint $X>N$ where N is the maximal value found so far.

Suppose that we have three kinds of items. Each item generates a profit but also imposes a "cost" (for instance space). The first item can be sold for 3 units of money and requires 2 units of space. The second item can be sold for 4 units and requires 3 units of space. The third item can be sold for 10 units but requires 7 units of space. How do we maximize the profit when the space is limited? The problem can be described easily using CLP(FD):

```
1. items(A,B,C,S,P) :-
2.     domain([A,B,C],0,10),
3.     AS #= 2*A, AP #= 3*A,
4.     BS #= 3*B, BP #= 4*B,
5.     CS #= 7*C, CP #= 10*C,
6.     S  #>= AS+BS+CS,
7.     P  #= AP+BP+CP,
8 .    labeling([maximize(P)],[P,S,A,B,C]).
```

Here `A`, `B` and `C` is the number of items of the three kinds. We assume (line 2) that there are at most 10 items of each kind. Line 3 describes the profit

and penalty generated by items of the first kind. Lines 4 and 5 introduce similar constraints for the other kinds of items. Line 6 bounds the total space consumption and line 7 defines the overall profit. In line 8 we generate the number of items of each kind together with the total space consumption while maximizing the profit. The program may be used as follows:

```
| ?- items(A, B, C, 13, P).

A = 3,
B = 0,
C = 1,
P = 19 ?
```

16.4 Global Constraints

Apart from primitive constraints involving arithmetic operations and comparisons many CLP(FD)-systems also contain more powerful constraints, often referred to as *global* constraints. The global constraints can generally be described, and may even be defined, by means of primitive constraints, but expressing the constraints may be cumbersome and, more seriously, lead to inefficiency; the global constraints are usually based on very efficient algorithms for checking e.g. satisfiability.

One of the simplest global constraints is the constraint `all_different`/1 – it is of the form:

$$\texttt{all\_different([}X_1, \ldots, X_n\texttt{])}$$

The constraint holds if $X_i \neq X_j$ whenever $i \neq j$. This can obviously be expressed by means of disequlities, but this would lead to a quadratic number of constraints – as a consequence some constraint systems employ special purpose algorithms to check satisfiability of the constraint. To illustrate a use of this constraint, consider the problem of assigning distinct integers in the domain $0 - 9$ to the letters S,E,N,D,M,O,R,Y so that the following equation holds:

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

(We also require that S and M are greater than zero.) The problem can be formalized as follows:

```
 1. smm([S,E,N,D,M,O,R,Y]) :-
 2.     domain([S,E,N,D,M,O,R,Y], 0, 9),
 3.     S #> 0, M #> 0,
 4.     all_different([S,E,N,D,M,O,R,Y]),
 5.     sum(S,E,N,D,M,O,R,Y),
 6.     labeling([], [S,E,N,D,M,O,R,Y]).

 7. sum(S, E, N, D, M, O, R, Y) :-
 8.                   1000*S + 100*E + 10*N + D
 9.                + 1000*M + 100*O + 10*R + E
10.    #= 10000*M + 1000*O + 100*N + 10*E + Y.
```

Line 2 defines the domains of the domain variables to $0 - 9$, and in line 3 the variables S and M are further restricted. Line 4 imposes the additional constraint that all variables should have distinct values. The predicate `sum`/1 holds if the the main equation holds and finally `labeling`/2 is used to enumerate all solutions:

```
| ?- smm([S,E,N,D,M,O,R,Y]).

D = 7, E = 5, M = 1, N = 6, O = 0, R = 8, S = 9, Y = 2
```

Note that even though there is only one solution to the constraints we need to use the predicate `labeling`/2 due to the incompleteness of the finite domain solver.

A class of problems that occur frequently in many applications is *resource allocation* problems. Imagine that we have a limited resource and collection of tasks that compete for the resources. Each task is characterized by its starting time S_i, the duration D_i and the number of resources R_i requested. The problem is to place the tasks so that the maximum number of resources is not exceeded at any point. This problem is embodied in the global constraint `cumulative`/4:

$$\texttt{cumulative}([S_1, \ldots, S_n], [D_1, \ldots, D_n], [R_1, \ldots, R_n], L)$$

The constraint holds if the total height of all rectangles at each integer-valued time-point is less than or equal to L.

The following constraints illustrate the use of `cumulative`/4. (One of the solutions is depicted in Figure 16.1.)

```
| ?- domain([S1,S2,S3],0,4),
     S1 #< S3,
     cumulative([S1,S2,S3],[3,4,2],[2,1,3],3),
```

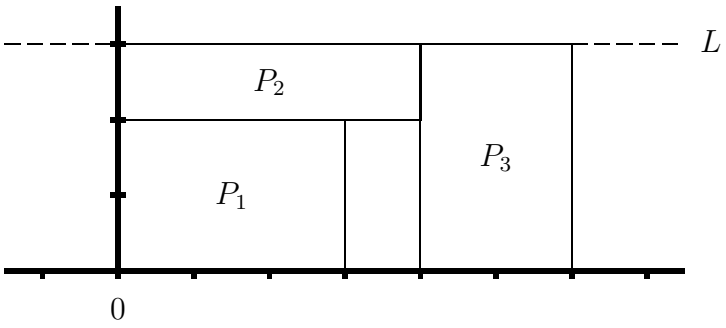

Figure 16.1: The cumulative constraint

```
labeling([],[S1,S2,S3]).

S1 = 0, S2 = 0, S3 = 4

S1 = 1, S2 = 0, S3 = 4
```

In the next example we consider a soccer-team with 11 players. The players spend the following amount of time in the showers:

Player	1	2	3	4	5	6	7	8	9	10	11
Time	5	3	8	2	7	3	9	3	3	5	7

What is the minimal time required for all players to come out of the showers if we assume that there are only three showers? The problem can be solved using the constraint `cumulative`/4. The limiting resource is the number of showers. Each player taking a shower is a "process" with a duration (specified above) and each process occupies one resource. The problem can be solved as follows:

```
 1. shower(S, Done) :-
 2.     D = [5, 3, 8, 2, 7, 3, 9, 3, 3, 5, 7],
 3.     R = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 4.     length(D, N),
 5.     length(S, N),
 6.     domain(S, 0, 100),
 7.     Done in 0..100,
 8.     ready(S, D, Done),
 9.     cumulative(S, D, R, 3),
10.     labeling([minimize(Done)], [Done|S]).
```

```
11. ready([], [], _).
12. ready([S|Ss], [D|Ds], Done) :-
13.     Done #>= S+D,
14.     ready(Ss, Ds, Done).
```

Line 2 specifies the durations of all processes and line 3 specifies the number of resources occupied by each process. Lines 4 to 6 are used to create 11 domain variables corresponding to the starting times of all processes. The domain of each variable is initially set to 0–100. Then we create a domain variable `Done` which will be used to bound the finishing time of all processes. Line 8 is used to specify that the finishing time (i.e. the starting time plus the duration) of each process is less than or equal to `Done`. Line 9 limits the number of resources to three and in line 10 we generate solutions for the starting times while minimizing the value of `Done`. A solution looks as follows:

```
| ?- shower(S,D).

D = 19,
S = [0,0,0,3,5,5,8,8,11,14,12]
```

The next constraint that we consider holds if Y is the X:th element of the list given as second element:

$$\texttt{element(}X\texttt{,[}X_1,\ldots,X_n\texttt{],}Y\texttt{)}$$

Here $X, Y, X_1, \ldots, X_n$ may be either integers or domain variables. A predicate `element(X,A,Y)` is similar to the equation `A[X] = Y` where `A` is an array.

To illustrate some uses of the constraint, consider the following:

```
| ?- element(X, [1,2,3,5], Y).
```

In this case the domain of `X` is 1–4 and the domain of `Y` is $\{1, 2, 3, 5\}$. Similarly:

```
| ?- X in 2..3, element(X, [1, X, 4, 5], Y).
```

In this case the domain of `X` is $\{2, 3\}$. The domain of `Y` is $\{2, 4\}$ in an optimal system (since `Y=2` if `X=2` and `Y=4` if `X=3`). However, most systems infer the domain 2–4 for `Y`.

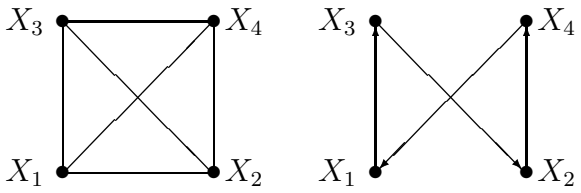

Figure 16.2: A graph and a subgraph which is a Hamiltonian cycle

The final constraint that we consider has applications in graph theory. Consider a graph with n nodes labeled $X_1, \ldots, X_n$ some of which are connected by edges. If we use domain variables to represent the nodes of a directed graph, we may represent the edges of the graph by constraining the domains of variables. For instance, if there are edges from X_1 to X_2 and X_3 we can specify this by the constraint `X1 in 2..3`. Now, a path that starts and ends in the same node and visits all other nodes exactly once is called a *Hamiltonian cycle*. For instance, the path X_1, X_3, X_2, X_4, X_1 is a Hamiltonian cycle in Figure 16.2. Finding a Hamiltonian cycle in a given graph amounts to removing all but one outgoing edge from each node while making sure that the remaining edges form a cycle involving all nodes. Now imagine that we represent going from X_i to X_j by assigning j to X_i. Then the following assignment encodes a Hamiltonian cycle involving the nodes $X_1 - X_4$:

$$X_1 \mapsto 3, X_2 \mapsto 4, X_3 \mapsto 2, X_4 \mapsto 1$$

This idea is embodied in the global constraint:

`circuit([`$X_1, \ldots, X_n$`])`

The constraint is true if the variables form a Hamiltonian cycle – that is, if each variable X_i $(1 \leq i \leq n)$ is assigned a unique integer in the domain $\{1, \ldots, n\}$ and if the assignments form a cycle (or a circuit). Note that if the domain variables are unbounded – that is, each node has an edge to all nodes including a self-referential edge – the constraint imposes the domain $\{1, \ldots, n\} - \{i\}$ on the variable X_i.

This constraint is useful for solving graph problems such as the traveling salesman problem. In the traveling salesman problem there are a number of cities. Each city is connected to all other cities and each connection has an associated cost (or distance). The problem is to find the cheapest tour that visits all towns exactly once and ends in the first city. (That is, a Hamiltonian cycle.)

Let us assume that there are seven cities with the following distances:

	X_1	X_2	X_3	X_4	X_5	X_6	X_7
X_1	–	4	8	10	7	14	15
X_2	4	–	7	7	10	12	5
X_3	8	7	–	4	6	8	10
X_4	10	7	4	–	2	5	8
X_5	7	10	6	2	–	6	7
X_6	14	12	8	5	6	–	5
X_7	15	5	10	8	7	5	–

The problem can be specified as follows using the global constraints `element`/3 and `circuit`/1:

```
 1. tsp(Cities, Cost) :-
 2.     Cities = [X1,X2,X3,X4,X5,X6,X7],
 3.     element(X1, [ 0, 4, 8,10, 7,14,15], C1),
 4.     element(X2, [ 4, 0, 7, 7,10,12, 5], C2),
 5.     element(X3, [ 8, 7, 0, 4, 6, 8,10], C3),
 6.     element(X4, [10, 7, 4, 0, 2, 5, 8], C4),
 7.     element(X5, [ 7,10, 6, 2, 0, 6, 7], C5),
 8.     element(X6, [14,12, 8, 5, 6, 0, 5], C6),
 9.     element(X7, [15, 5,10, 8, 7, 5, 0], C7),
10.     Cost #= C1+C2+C3+C4+C5+C6+C7,
11.     circuit(Cities),
12.     labeling([minimize(Cost)], Cities).
```

We first create domain variables corresponding to the cities that we have to visit exactly once. We also, in line 3-9, create domain variables describing the cost of going from one city to the next. Lines 3 to 9 are also responsible for defining the domains of the variables – for instance, line 3 defines the domain of `X1` to be $1 - 7$ and the domain of `C1` to be $\{0, 4, 7, 8, 10, 14, 15\}$ – the possible cost of going from `X1` to the next city.

Line 10 introduces yet another domain variable which is set to the total cost of visiting all cities. We then say that the tour should be a Hamiltonian circuit and finally we solve all the constraints while minimizing the total cost. In this particular example the answer looks as follows:

```
| ?- tsp(X,C).

C = 34,
X = [2,7,1,3,4,5,6] ?
```

That is, we go from $X_1 \mapsto X_2 \mapsto X_7 \mapsto X_6 \mapsto X_5 \mapsto X_4 \mapsto X_3 \mapsto X_1$ with a total cost of 34.