Constraint Logic Programming: a Brief Introduction

In CLP, a number of constraint solvers can be embedded into Prolog, including solvers over reals, rationals, booleans, etc. Perhaps the most useful one is over *finite domains*. This is typically based on the general *Constraint Logic Programming Scheme* introduced by Jaffar and Michaylov in 1987. The idea is this: unification in Prolog is about solving constraints over equations in the sense of pattern matching. Now, we may embed an arbitrary constraint solver in the place of unification to solve not only constraints over equations, but constraints over other domains.

Take a look at a simple example. Suppose we write a Prolog program:

p(f(X)) :- X>1, q(X).  
 q(X) :- X<5.

If we call

?- p(Z).

we will get the complaint from Prolog that X is uninstantiated when trying to solve X > 1. However, if we consider whether X > 1 is solvable over real numbers, the answer is obvious: YES. In this case, we are not interested in bindings of X (there are infinitely many), but only the question whether it is satisfiable (solvable).

Since we may have a conjunction of equations and inequalities to solve, let's leave it to a *constraint store*, a collection of some primitive constraints (those composed of comparison operators such as X > 1). Any non-user defined predicates will be decomposed into primitive constraints which are sent to the constraint store. Every time a constraint store is modified, we check, and if we can determine that it cannot be solved we can then stop right away. Then, Prolog backtracks in the normal way.

We can sketch how this is done below where CS is the constraint store at each step.

solve p(Z) CS = {}  
 |   
 V  
 X > 1, q(X) CS = {Z = f(X)}  
 |   
 V  
 q(X) CS = {Z = f(X), X > 1}  
 |  
 V   
 X < 5 CS = {Z = f(X), X > 1}  
 |  
 V CS = {Z = f(X), X > 1, X < 5}  
 []

Now the answer to the original query lies in whether CS is sovable, which we know is yes. So if a solver over equations and inequalities is embedded in Prolog, we will get a yes answer.

In any Prolog implementation, you have to use some specific syntax in order to invoke an embedded constraint solver. In SWI Prolog, you first load the clpr library, and write your program as:

:- use\_module(library(clpr)).   
  
p(f(X)) :- {X>1}, q(X).  
q(X) :- {X<5}.

The first line loads the library for the constraint solver over reals (CLPR). The braces around an inequality or an equation tell SWI Prolog that the inequality should be sent to the constraint store and solved by CLPR. Now, if you type the goal

?- p(Z)

you will get something like

Z = f(\_A),  
 {\_A<5.0},  
 {\_A>1.0}

The last two lines indicate that these inequalities are solvable.

**Example.**

:- use\_module(library(clpr)).   
  
p(f(X)) :- {X>1}, q(X).  
q(X) :- {X<0}, g(X).  
g(1).

solve p(Z) CS = {}  
 |   
 V  
 X > 1, q(X) CS = {Z = f(X)}  
 |   
 V  
 q(X) CS = {Z = f(X), X > 1}  
 |  
 V   
 X < 0, g(X) CS = {Z = f(X), X > 1}  
 |  
 V CS = {Z = f(X), X > 1, X < 0}  
 g(X)

In this case, the proof fails, since CS is not solvable.

## Solving CSPs using the SWI Prolog Finite Domain Solver

This section is based on the [manual](http://www.swi-prolog.org/pldoc/man?section=clpfd) by Markus Triska, and on the examples on [his github page](https://github.com/triska/clpfd).

In this course, we will be using a CLP(FD) solver for solving constraint logic problems (CLP) over finite domains (FD). It comes with SWI Prolog as a library. This implementation of CLPFD extends Prolog to allow many kind of relations over variables, and can be used to solve CSPs efficiently. It is claimed that about 90% of industrial applications are finite domain constraint problems. The constraint solver is useful for modelling problems such as scheduling, planning, packing, timetabling, etc.

A CSP, is described by three things: the variables, their domains, and the constraints. The overall structure for using CLP(FD) with SWI Prolog is:

* Load the library with use\_module(library(clpfd)).
* Specify constraints
* Optionally, declare ranges of values for your variables
* tell Prolog to use *labeling*: a systematic search over the remaining possible values for the variables, until a solution that satisfies all constraints is found, or until it is proven that no solution exists.

Writing constraint programs is very much like writing a normal Prolog program, with a specific syntax to add constraints and a labeling predicate to start the search.

**Specifying constraints**

Constraints to be added to a constraint store are prefixed with a #, to distinguish them from the normal Prolog relational operators. The comparisons are #= (equality), #\= (inequality), #<, #=<, #>, #>=.

[Example](https://github.com/triska/clpfd/blob/master/sendmory.pl) of SEND+MORE=MONEY, from Triska:

To specify that the leading digits M and S can not be 0, we can use constraints:

M #\= 0, S #\= 0.

To specify the equation as a constraint using #= :

S\*1000 + E\*100 + N\*10 + D +

M\*1000 + O\*100 + R\*10 + E #=

M\*10000 + O\*1000 + N\*100 + E\*10 + Y.

Note that both sides of a constraint can be arithmetic expressions. This goes far beyond the usual Prolog capabilities.

**Restricting domains**

In SWI Prolog, you can use the builtin predicates *in/2 and ins/2* to restrict variables to domains which are subsets of all integers, usually given as one or more ranges. Ranges are specified using the notation a..b, which means all integers x with a <= x <= b. Several ranges can be combined using the \/ operator. Examples:

X in 1..4 restricts X to possible values 1, 2, 3, or 4.

[X,Y,Z] ins 1..4 restricts all three variables to 1..4.

X in 1..3\/9..12 restricts X to 1, 2, 3, 9, 10, 11, or 12.

The special symbols inf and sup stand for -∞ and +∞, respectively.

To come back to SEND+MORE=MONEY, we can express that the variables are digits with a range of 0..9 as follows:

Vars = [S,E,N,D,M,O,R,Y],  
Vars ins 0..9,

Finally, we need to express that all variables stand for different digits. we could either add choose(8,2)=28 constraints such as S #\=E, S #\= N, ... by hand, or (better and clearer) use the builtin predicate *all\_different/1* or *all\_distinct/1*:

all\_different(Vars).

At this point, it is interesting to see how far CLP(FD) can reduce the variable domains. Here is the complete puzzle specification from Triska, with constraints and domains but without search:

:- use\_module(library(clpfd)).  
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-  
        Vars = [S,E,N,D,M,O,R,Y],  
        Vars ins 0..9,  
        all\_different(Vars),  
                  S\*1000 + E\*100 + N\*10 + D +  
                  M\*1000 + O\*100 + R\*10 + E #=  
        M\*10000 + O\*1000 + N\*100 + E\*10 + Y,  
        M #\= 0, S #\= 0.

We can run this directly in SWI Prolog:

?- use\_module(library(clpfd)).  
true.  
?- [user].  
puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-  
|:         Vars = [S,E,N,D,M,O,R,Y],  
|:         Vars ins 0..9,  
|:         all\_different(Vars),  
|:                   S\*1000 + E\*100 + N\*10 + D +  
|:                   M\*1000 + O\*100 + R\*10 + E #=  
|:         M\*10000 + O\*1000 + N\*100 + E\*10 + Y,  
|:         M #\= 0, S #\= 0.  
|: % user://1 compiled 0.01 sec, 2 clauses  
true.  
?- puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]).  
S = 9,  
M = 1,  
O = 0,  
E in 4..7,  
all\_different([9, E, N, D, 1, 0, R, Y]),  
1000\*9+91\*E+ -90\*N+D+ -9000\*1+ -900\*0+10\*R+ -1\*Y#=0,  
N in 5..8,  
D in 2..8,  
R in 2..8,  
Y in 2..8.

By just using constraint propagation, **no search at all**, CLP(FD) has already figured out some variable values, and greatly reduced the domains of all others.

**Searching for Solutions**

To tell the CLP(FD) solver to assign a domain value to a variable, one by one, until a satisfying assignment is found, you use *label/1*or*labeling/2.*

label([X,Y,Z])

assigns possible values to X, Y, Z, in that order, one by one, until a solution is found. The first parameter of labeling/2, is used to specify search options. E.g. you can choose the order from Z to X rather than from X to Z, or choose the variable that has the smallest domain size. Leaving it empty [], means using the default options as in *label/1*.

So now, we can complete our puzzle with the following query:

?- puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]), label([S,E,N,D,M,O,R,Y]).  
S = 9,  
E = 5,  
N = 6,  
D = 7,  
M = 1,  
O = 0,  
R = 8,  
Y = 2 .

It is also interesting to only label (systematically search for) some variables, and see if constraint propagation can solve the rest or not:

?- puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]), label([E]).  
S = 9,  
E = 5,  
N = 6,  
D = 7,  
M = 1,  
O = 0,  
R = 8,  
Y = 2 .

?- puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]), label([Y]).  
S = 9,  
M = 1,  
O = 0,  
Y = 2,  
E in 4..7,  
all\_different([9, E, N, D, 1, 0, R, 2]),  
1000\*9+91\*E+ -90\*N+D+ -9000\*1+ -900\*0+10\*R+ -1\*2#=0,  
N in 5..8,  
D in 3..8,  
R in 3..8 .

**Search Control Options**

The first parameter in *labeling/2* is a list of *options* for search control. The most common search control is the order in which the next variable is selected for assignment. That is, for the strategy of next variable selection, you place one of the following into the list.

**leftmost**

The leftmost variable is selected. This is the default.

**min**

The leftmost variable with the smallest lower bound is selected.

**max**

The leftmost variable with the greatest upper bound is selected.

**ff**

The first-fail principle is used: the leftmost variable with the smallest domain is selected.

**ffc**

The most constrained heuristic is used: a variable with the smallest domain is selected, breaking ties by (a) selecting the variable that has the most constraints suspended on it and (b) selecting the leftmost one.

Once you have a variable selected, you may want to fix the order in which values in the domain are selected.

**up**

The domain is explored in ascending order. This is the default.

**down**

The domain is explored in descending order.

It is important to understand how labeling works. Consider this program:

%--------------------------------------------------------------------  
% Without labeling, domain reductions are performed but   
% you may not get a fully instantiated solution  
  
noLabeling(N,L) :-  
 L = [A,B,C],  
 L ins 1..N,   
 A #> B, B #> C.  
  
?- noLabeling(4,L).

L = [\_G2654, \_G2657, \_G2660],

\_G2654 in 3..4,

\_G2657#=<\_G2654+ -1,

\_G2657 in 2..3,

\_G2660#=<\_G2657+ -1,

\_G2660 in 1..2.

The domains for all three variables have been reduced by applying arc consistency, and the constraints are kept. Since no labeling is used, variables did not get a specific value.

The predicate labeling simply relies on Prolog's backtracking to get domain instantiate variables. Here, indomain/1 is a CLPFD built-in predicate that assigns domain values to variables in increasing order via backtracking. To understand it, we can write our own labeling predicate as follows:

mylabel([]).  
mylabel([V|Vs]) :- indomain(V), mylabel(Vs).  
goal(X,Y) :- Vars = [X,Y], X in 10..11, Y in 1..2\/5..6, mylabel(Vars).

Now we can see the backtracking at work:

?- goal(X,Y).

X = 10,

Y = 1 ;

X = 10,

Y = 2 ;

X = 10,

Y = 5 ;

X = 10,

Y = 6 ;

X = 11,

Y = 1 ;

X = 11,

Y = 2 ;

X = 11,

Y = 5 ;

X = 11,

Y = 6.

mylabel generates all valid combinations of X and Y by backtracking.  
  
Now, mylabel with constraints:

?- [user].

t(X,Y,Z) :-

|:    Vars = [X,Y,Z], Vars ins 1..4,

|:    X#>Y, Y#>Z,

|:    mylabel([X,Y,Z]).

|: % user://5 compiled 0.00 sec, 2 clauses

true.

?- t(X,Y,Z).

X = 3,

Y = 2,

Z = 1 ;

X = 4,

Y = 2,

Z = 1 ;

X = 4,

Y = 3,

Z = 1 ;

X = 4,

Y = 3,

Z = 2.

Here are some interesting programs, again from Triska's page:

* [factorial](https://github.com/triska/clpfd/blob/master/n_factorial.pl)
* [n-queens](https://github.com/triska/clpfd/blob/master/n_queens.pl)
* [Sudoku](https://github.com/triska/clpfd/blob/master/sudoku.pl)

Last modified: Monday, 21 March 2016, 10:04 PM