A.7 library(clpfd): Constraint Logic Programming over Finite Domains

author
Markus Triska

Constraint programming is a declarative formalism that lets you describe conditions a solution must satisfy. This library provides CLP(FD), Constraint Logic Programming over Finite Domains. It can be used to model and solve various combinatorial problems such as planning, scheduling and allocation tasks.

Most predicates of this library are finite domain *constraints*, which are relations over integers. They generalise arithmetic evaluation of integer expressions in that propagation can proceed in all directions. This library also provides *enumeration predicates*, which let you systematically search for solutions on variables whose domains have become finite. A finite domain *expression* is one of:

an integer	Given value
a variable	Unknown value
-Expr	Unary minus
Expr + Expr	Addition
Expr * Expr	Multiplication
Expr - Expr	Subtraction
Expr ^ Expr	Exponentiation
min(Expr,Expr)	Minimum of two expressions
max(Expr,Expr)	Maximum of two expressions
Expr mod Expr	Modulo induced by floored division
Expr rem Expr	Modulo induced by truncated division
abs(Expr)	Absolute value
Expr / Expr	Truncated integer division

The most important finite domain constraints are:

Expr1 #>= Expr2	Expr1 is larger than or equal to Expr2
Expr1 #=< Expr2	Expr1 is smaller than or equal to Expr2
Expr1 #= Expr2	Expr1 equals Expr2
Expr1 #\= Expr2	Expr1 is not equal to Expr2
Expr1 #> Expr2	Expr1 is strictly larger than Expr2
Expr1 #< Expr2	Expr1 is strictly smaller than Expr2

The constraints in/2, #=/2, #\=/2, #</2, #>/2, #=</2, and #>=/2 can be *reified*, which means reflecting their truth values into Boolean values represented by the integers 0 and 1. Let P and Q denote reifiable constraints or Boolean variables, then:

#\ Q	True iff Q is false
P #\/ Q	True iff either P or Q
P #/\ Q	True iff both P and Q
P #<==> Q	True iff P and Q are equivalent
P #==> Q	True iff P implies Q
P #<== Q	True iff Q implies P

The constraints of this table are reifiable as well. If a variable occurs at the place of a constraint that is being reified, it is implicitly constrained to the Boolean values 0 and 1. Therefore, the following queries all fail: `?- #\ 2.`, `?- #\ #\ 2.` etc.

Here is an example session with a few queries and their answers:

```
?- [library(clpfd)].
% library(clpfd) compiled into clpfd 0.06 sec, 3,308 bytes
```

Home
Download
Browse GIT
Contrib
Packs

HOWTO
FAQ
Manual
Mailinglist
Support
Links
Contact

IDE (developing)
Graphics
(Semantic) Web

Publications
Contributors
License

Wiki (edit)

```
true.

?- X #> 3.
X in 4..sup.

?- X #\= 20.
X in inf..19\/21..sup.

?- 2*X #= 10.
X = 5.

?- X*X #= 144.
X in -12\/12.

?- 4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
X = 3,
Y = 6.

?- Vs = [X,Y,Z], Vs ins 1..3, all_different(Vs), X = 1, Y #\= 2.
Vs = [1, 3, 2],
X = 1,
Y = 3,
Z = 2.

?- X #= Y #<==> B, X in 0..3, Y in 4..5.
B = 0,
X in 0..3,
Y in 4..5.
```

In each case (and as for all pure programs), the answer is declaratively equivalent to the original query, and in many cases the constraint solver has deduced additional domain restrictions.

A common usage of this library is to first post the desired constraints among the variables of a model, and then to use enumeration predicates to search for solutions. As an example of a constraint satisfaction problem, consider the cryptoarithmetic puzzle SEND + MORE = MONEY, where different letters denote distinct integers between 0 and 9. It can be modeled in CLP(FD) as follows:

```
:- 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.
```

Sample query and its result (actual variables replaced for readability):

```
?- puzzle(As+Bs=Cs).
As = [9, _A2, _A3, _A4],
Bs = [1, 0, _B3, _A2],
Cs = [1, 0, _A3, _A2, _C5],
_A2 in 4..7,
all_different([9, _A2, _A3, _A4, 1, 0, _B3, _C5]),
1000*9+91*_A2+ -90*_A3+_A4+ -9000*1+ -900*0+10*_B3+ -1*_C5#=0,
_A3 in 5..8,
_A4 in 2..8,
_B3 in 2..8,
_C5 in 2..8.
```

Here, the constraint solver has deduced more stringent bounds for all variables. Keeping the modeling part separate from the search lets you view these residual goals, observe termination and determinism properties of the modeling part in isolation from the search, and more easily experiment with different search strategies. Labeling can then be used to search for solutions:

```
?- puzzle(As+Bs=Cs), label(As).
As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2] ;
false.
```

In this case, it suffices to label a subset of variables to find the puzzle's unique solution, since the constraint

solver is strong enough to reduce the domains of remaining variables to singleton sets. In general though, it is necessary to label all variables to obtain ground solutions.

You can also use CLP(FD) constraints as a more declarative alternative for ordinary integer arithmetic with is/2, >/2 etc. For example:

```
:- use_module(library(clpfd)).

n_factorial(0, 1).
n_factorial(N, F) :-
        N #> 0, N1 #= N - 1, F #= N * F1,
        n_factorial(N1, F1).
```

This predicate can be used in all directions. For example:

```
?- n_factorial(47, F).
F = 258623241511168180642964355153611979969197632389120000000000 ;
false.

?- n_factorial(N, 1).
N = 0 ;
N = 1 ;
false.

?- n_factorial(N, 3).
false.
```

To make the predicate terminate if any argument is instantiated, add the (implied) constraint F #\= 0 before the recursive call. Otherwise, the query n_factorial(N, 0) is the only non-terminating case of this kind.

This library uses goal_expansion/2 to rewrite constraints at compilation time. The expansion's aim is to transparently bring the performance of CLP(FD) constraints close to that of conventional arithmetic predicates (</2, =:=/2, is/2 etc.) when the constraints are used in modes that can also be handled by built-in arithmetic. To disable the expansion, set the flag clpfd_goal_expansion to false.

Use call_residue_vars/2 and copy_term/3 to inspect residual goals and the constraints in which a variable is involved. This library also provides *reflection* predicates (like fd_dom/2, fd_size/2 etc.) with which you can inspect a variable's current domain. These predicates can be useful if you want to implement your own labeling strategies.

You can also define custom constraints. The mechanism to do this is not yet finalised, and we welcome suggestions and descriptions of use cases that are important to you. As an example of how it can be done currently, let us define a new custom constraint "oneground(X,Y,Z)", where Z shall be 1 if at least one of X and Y is instantiated:

```
:- use_module(library(clpfd)).

:- multifile clpfd:run_propagator/2.

oneground(X, Y, Z) :-
        clpfd:make_propagator(oneground(X, Y, Z), Prop),
        clpfd:init_propagator(X, Prop),
        clpfd:init_propagator(Y, Prop),
        clpfd:trigger_once(Prop).

clpfd:run_propagator(oneground(X, Y, Z), MState) :-
        (   integer(X) -> clpfd:kill(MState), Z = 1
        ;   integer(Y) -> clpfd:kill(MState), Z = 1
        ;   true
        ).
```

First, **clpfd:make_propagator/2** is used to transform a user-defined representation of the new constraint to an internal form. With **clpfd:init_propagator/2**, this internal form is then attached to X and Y. From now on, the propagator will be invoked whenever the domains of X or Y are changed. Then, **clpfd:trigger_once/1** is used to give the propagator its first chance for propagation even though the variables' domains have not yet changed. Finally, **clpfd:run_propagator/2** is extended to define the actual propagator. As explained, this predicate is automatically called by the constraint solver. The first argument is the user-defined representation of the constraint as used in **clpfd:make_propagator/2**, and the second argument is a mutable state that can be used to prevent further invocations of the propagator when the constraint has become entailed, by using **clpfd:kill/1**. An example of using the new constraint:

```
?- oneground(X, Y, Z), Y = 5.
Y = 5,
Z = 1,
X in inf..sup.
```

You can cite this library in your publications as:

```
@inproceedings{Triska12,
  author    = {Markus Triska},
  title     = {The Finite Domain Constraint Solver of {SWI-Prolog}},
  booktitle = {FLOPS},
  series    = {LNCS},
  volume    = {7294},
  year      = {2012},
  pages     = {307-316}
}
```

?Var **in** *+Domain*

Var is an element of *Domain*. *Domain* is one of:

Integer

Singleton set consisting only of *Integer*.

Lower **..** *Upper*

All integers *I* such that *Lower* =< *I* =< *Upper*. *Lower* must be an integer or the atom **inf**, which denotes negative infinity. *Upper* must be an integer or the atom **sup**, which denotes positive infinity.

Domain1 \/ *Domain2*

The union of Domain1 and Domain2.

+Vars **ins** *+Domain*

The variables in the list *Vars* are elements of *Domain*.

indomain(*?Var*)

Bind *Var* to all feasible values of its domain on backtracking. The domain of *Var* must be finite.

label(*+Vars*)

Equivalent to labeling([], *Vars*).

labeling(*+Options*, *+Vars*)

Assign a value to each variable in *Vars*. Labeling means systematically trying out values for the finite domain variables *Vars* until all of them are ground. The domain of each variable in *Vars* must be finite. *Options* is a list of options that let you exhibit some control over the search process. Several categories of options exist:

The variable selection strategy lets you specify which variable of *Vars* is labeled next and is one of:

leftmost

Label the variables in the order they occur in *Vars*. This is the default.

ff

First fail. Label the leftmost variable with smallest domain next, in order to detect infeasibility early. This is often a good strategy.

ffc

Of the variables with smallest domains, the leftmost one participating in most constraints is labeled next.

min

Label the leftmost variable whose lower bound is the lowest next.

max

Label the leftmost variable whose upper bound is the highest next.

The value order is one of:

up

Try the elements of the chosen variable's domain in ascending order. This is the default.

down

Try the domain elements in descending order.

The branching strategy is one of:

step
For each variable X, a choice is made between X = V and X #\= V, where V is determined by the value ordering options. This is the default.

enum
For each variable X, a choice is made between X = V_1, X = V_2 etc., for all values V_i of the domain of X. The order is determined by the value ordering options.

bisect
For each variable X, a choice is made between X #=< M and X #> M, where M is the midpoint of the domain of X.

At most one option of each category can be specified, and an option must not occur repeatedly.

The order of solutions can be influenced with:

min(*Expr*)
max(*Expr*)

This generates solutions in ascending/descending order with respect to the evaluation of the arithmetic expression Expr. Labeling *Vars* must make Expr ground. If several such options are specified, they are interpreted from left to right, e.g.:

```
?- [X,Y] ins 10..20, labeling([max(X),min(Y)],[X,Y]).
```

This generates solutions in descending order of X, and for each binding of X, solutions are generated in ascending order of Y. To obtain the incomplete behaviour that other systems exhibit with "maximize(Expr)" and "minimize(Expr)", use once/1, e.g.:

```
once(labeling([max(Expr)], Vars))
```

Labeling is always complete, always terminates, and yields no redundant solutions.

all_different(+*Vars*)
Vars are pairwise distinct.

all_distinct(+*Ls*)
Like all_different/1, with stronger propagation. For example, all_distinct/1 can detect that not all variables can assume distinct values given the following domains:

```
?- maplist(in, V,
           [1\/3..4, 1..2\/4, 1..2\/4, 1..3, 1..3, 1..6]),
   all_distinct(V).
false.
```

sum(+*Vars*, +*Rel*, *?Expr*)
The sum of elements of the list *Vars* is in relation *Rel* to *Expr*. *Rel* is one of #=, #\=, #<, #>, #=< or #>=. For example:

```
?- [A,B,C] ins 0..sup, sum([A,B,C], #=, 100).
A in 0..100,
A+B+C#=100,
B in 0..100,
C in 0..100.
```

scalar_product(+*Cs*, +*Vs*, +*Rel*, *?Expr*)
Cs is a list of integers, *Vs* is a list of variables and integers. True if the scalar product of *Cs* and *Vs* is in relation *Rel* to *Expr*, where *Rel* is #=, #\=, #<, #>, #=< or #>=.

?X #>= *?Y*
X is greater than or equal to *Y*.

?X #=< *?Y*
X is less than or equal to *Y*.

?X #= *?Y*

X equals *Y*.

?X #\= *?Y*
X is not *Y*.

?X #> *?Y*
X is greater than *Y*.

?X #< *?Y*
X is less than *Y*. In addition to its regular use in problems that require it, this constraint can also be useful to eliminate uninteresting symmetries from a problem. For example, all possible matches between pairs built from four players in total:

```
?- Vs = [A,B,C,D], Vs ins 1..4,
        all_different(Vs),
        A #< B, C #< D, A #< C,
   findall(pair(A,B)-pair(C,D), label(Vs), Ms).
Ms = [ pair(1, 2)-pair(3, 4),
       pair(1, 3)-pair(2, 4),
       pair(1, 4)-pair(2, 3)].
```

#\ *+Q*
The reifiable constraint *Q* does *not* hold. For example, to obtain the complement of a domain:

```
?- #\ X in -3..0\/10..80.
X in inf.. -4\/1..9\/81..sup.
```

?P #<==> *?Q*
P and *Q* are equivalent. For example:

```
?- X #= 4 #<==> B, X #\= 4.
B = 0,
X in inf..3\/5..sup.
```

The following example uses reified constraints to relate a list of finite domain variables to the number of occurrences of a given value:

```
:- use_module(library(clpfd)).

vs_n_num(Vs, N, Num) :-
        maplist(eq_b(N), Vs, Bs),
        sum(Bs, #=, Num).

eq_b(X, Y, B) :- X #= Y #<==> B.
```

Sample queries and their results:

```
?- Vs = [X,Y,Z], Vs ins 0..1, vs_n_num(Vs, 4, Num).
Vs = [X, Y, Z],
Num = 0,
X in 0..1,
Y in 0..1,
Z in 0..1.

?- vs_n_num([X,Y,Z], 2, 3).
X = 2,
Y = 2,
Z = 2.
```

?P #==> *?Q*
P implies *Q*.

?P #<== *?Q*
Q implies *P*.

?P #/\ *?Q*
P and *Q* hold.

?P #\/ *?Q*
P or *Q* holds. For example, the sum of natural numbers below 1000 that are multiples of 3 or 5:

```
?- findall(N, (N mod 3 #= 0 #\/ N mod 5 #= 0, N in 0..999,
               indomain(N)),
           Ns),
   sum(Ns, #=, Sum).
Ns = [0, 3, 5, 6, 9, 10, 12, 15, 18|...],
Sum = 233168.
```

lex_chain(*+Lists*)
Lists are lexicographically non-decreasing.

tuples_in(*+Tuples, +Relation*)
Relation must be a list of lists of integers. The elements of the list *Tuples* are constrained to be elements of *Relation*. Arbitrary finite relations, such as compatibility tables, can be modeled in this way. For example, if 1 is compatible with 2 and 5, and 4 is compatible with 0 and 3:

```
?- tuples_in([[X,Y]], [[1,2],[1,5],[4,0],[4,3]]), X = 4.
X = 4,
Y in 0\/3.
```

As another example, consider a train schedule represented as a list of quadruples, denoting departure and arrival places and times for each train. In the following program, Ps is a feasible journey of length 3 from A to D via trains that are part of the given schedule.

```
:- use_module(library(clpfd)).

trains([[1,2,0,1],
        [2,3,4,5],
        [2,3,0,1],
        [3,4,5,6],
        [3,4,2,3],
        [3,4,8,9]]).

threepath(A, D, Ps) :-
        Ps = [[A,B,_T0,T1],[B,C,T2,T3],[C,D,T4,_T5]],
        T2 #> T1,
        T4 #> T3,
        trains(Ts),
        tuples_in(Ps, Ts).
```

In this example, the unique solution is found without labeling:

```
?- threepath(1, 4, Ps).
Ps = [[1, 2, 0, 1], [2, 3, 4, 5], [3, 4, 8, 9]].
```

serialized(*+Starts, +Durations*)
Constrain a set of intervals to a non-overlapping sequence. *Starts* = [S_1,...,S_n], is a list of variables or integers, *Durations* = [D_1,...,D_n] is a list of non-negative integers. Constrains *Starts* and *Durations* to denote a set of non-overlapping tasks, i.e.: S_i + D_i =< S_j or S_j + D_j =< S_i for all 1 =< i < j =< n. Example:

```
?- length(Vs, 3),
   Vs ins 0..3,
   serialized(Vs, [1,2,3]),
   label(Vs).
Vs = [0, 1, 3] ;
Vs = [2, 0, 3] ;
false.
```

See also
Dorndorf et al. 2000, "Constraint Propagation Techniques for the Disjunctive Scheduling Problem"

element(*?N, +Vs, ?V*)
The *N*-th element of the list of finite domain variables *Vs* is *V*. Analogous to nth1/3.

global_cardinality(*+Vs, +Pairs*)
Global Cardinality constraint. Equivalent to global_cardinality(*Vs*, *Pairs*, []). Example:

```
?- Vs = [_,_,_], global_cardinality(Vs, [1-2,3-_]), label(Vs).
Vs = [1, 1, 3] ;
Vs = [1, 3, 1] ;
Vs = [3, 1, 1].
```

global_cardinality(*+Vs, +Pairs, +Options*)

Global Cardinality constraint. *Vs* is a list of finite domain variables, *Pairs* is a list of Key-Num pairs, where Key is an integer and Num is a finite domain variable. The constraint holds iff each V in *Vs* is equal to some key, and for each Key-Num pair in *Pairs*, the number of occurrences of Key in *Vs* is Num. *Options* is a list of options. Supported options are:

consistency(*value*)

A weaker form of consistency is used.

cost(*Cost*, *Matrix*)

Matrix is a list of rows, one for each variable, in the order they occur in *Vs*. Each of these rows is a list of integers, one for each key, in the order these keys occur in *Pairs*. When variable v_i is assigned the value of key k_j, then the associated cost is Matrix_{ij}. Cost is the sum of all costs.

circuit(*+Vs*)

True if the list *Vs* of finite domain variables induces a Hamiltonian circuit. The k-th element of *Vs* denotes the successor of node k. Node indexing starts with 1. Examples:

```
?- length(Vs, _), circuit(Vs), label(Vs).
Vs = [] ;
Vs = [1] ;
Vs = [2, 1] ;
Vs = [2, 3, 1] ;
Vs = [3, 1, 2] ;
Vs = [2, 3, 4, 1] .
```

cumulative(*+Tasks*)

Equivalent to cumulative(*Tasks*, [limit(1)]).

cumulative(*+Tasks, +Options*)

Tasks is a list of tasks, each of the form task(S_i, D_i, E_i, C_i, T_i). S_i denotes the start time, D_i the positive duration, E_i the end time, C_i the non-negative resource consumption, and T_i the task identifier. Each of these arguments must be a finite domain variable with bounded domain, or an integer. The constraint holds if at any time during the start and end of each task, the total resource consumption of all tasks running at that time does not exceed the global resource limit (which is 1 by default). *Options* is a list of options. Currently, the only supported option is:

limit(*L*)

The integer L is the global resource limit.

automaton(*+Signature, +Nodes, +Arcs*)

Constrain variables with a finite automaton. Equivalent to automaton(_, _, *Signature*, *Nodes*, *Arcs*, [], [], _), a common use case of automaton/8. In the following example, a list of binary finite domain variables is constrained to contain at least two consecutive ones:

```
:- use_module(library(clpfd)).

two_consecutive_ones(Vs) :-
        automaton(Vs, [source(a),sink(c)],
                  [arc(a,0,a), arc(a,1,b),
                   arc(b,0,a), arc(b,1,c),
                   arc(c,0,c), arc(c,1,c)]).

?- length(Vs, 3), two_consecutive_ones(Vs), label(Vs).
Vs = [0, 1, 1] ;
Vs = [1, 1, 0] ;
Vs = [1, 1, 1].
```

automaton(*?Sequence, ?Template, +Signature, +Nodes, +Arcs, +Counters, +Initials, ?Finals*)

Constrain variables with a finite automaton. True if the finite automaton induced by *Nodes* and *Arcs* (extended with *Counters*) accepts *Signature*. *Sequence* is a list of terms, all of the same shape. Additional constraints must link *Sequence* to *Signature*, if necessary. *Nodes* is a list of source(Node) and sink(Node) terms. *Arcs* is a list of arc(Node,Integer,Node) and arc(Node,Integer,Node,Exprs) terms that denote the automaton's transitions. Each node is represented by an arbitrary term. Transitions that are not mentioned go to an implicit failure node. Exprs is a list of arithmetic expressions, of the same length as *Counters*. In each expression, variables occurring in *Counters* correspond to old counter values, and variables occurring in *Template* correspond to the current element of *Sequence*. When a transition containing expressions is taken, counters are updated as stated. By default, counters remain unchanged. *Counters* is a list of variables that must not occur

anywhere outside of the constraint goal. *Initials* is a list of the same length as *Counters*. Counter arithmetic on the transitions relates the counter values in *Initials* to *Finals*.

The following example is taken from Beldiceanu, Carlsson, Debruyne and Petit: "Reformulation of Global Constraints Based on Constraints Checkers", Constraints 10(4), pp 339-362 (2005). It relates a sequence of integers and finite domain variables to its number of inflexions, which are switches between strictly ascending and strictly descending subsequences:

```
:- use_module(library(clpfd)).

sequence_inflexions(Vs, N) :-
        variables_signature(Vs, Sigs),
        automaton(_, _, Sigs,
                  [source(s),sink(i),sink(j),sink(s)],
                  [arc(s,0,s), arc(s,1,j), arc(s,2,i),
                   arc(i,0,i), arc(i,1,j,[C+1]), arc(i,2,i),
                   arc(j,0,j), arc(j,1,j),
                   arc(j,2,i,[C+1])],
                  [C], [0], [N]).

variables_signature([], []).
variables_signature([V|Vs], Sigs) :-
        variables_signature_(Vs, V, Sigs).

variables_signature_([], _, []).
variables_signature_([V|Vs], Prev, [S|Sigs]) :-
        V #= Prev #<==> S #= 0,
        Prev #< V #<==> S #= 1,
        Prev #> V #<==> S #= 2,
        variables_signature_(Vs, V, Sigs).
```

Example queries:

```
?- sequence_inflexions([1,2,3,3,2,1,3,0], N).
N = 3.

?- length(Ls, 5), Ls ins 0..1,
   sequence_inflexions(Ls, 3), label(Ls).
Ls = [0, 1, 0, 1, 0] ;
Ls = [1, 0, 1, 0, 1].
```

transpose(*+Matrix*, *?Transpose*)

Transpose a list of lists of the same length. Example:

```
?- transpose([[1,2,3],[4,5,6],[7,8,9]], Ts).
Ts = [[1, 4, 7], [2, 5, 8], [3, 6, 9]].
```

zcompare(*?Order*, *?A*, *?B*)

Analogous to compare/3, with finite domain variables *A* and *B*. Example:

```
:- use_module(library(clpfd)).

 n_factorial(N, F) :-
         zcompare(C, N, 0),
         n_factorial_(C, N, F).

 n_factorial_(=, _, 1).
 n_factorial_(>, N, F) :-
      F #= F0*N, N1 #= N - 1,
      n_factorial(N1, F0).
```

This version is deterministic if the first argument is instantiated:

```
?- n_factorial(30, F).
F = 265252859812191058636308480000000.
```

chain(*+Zs*, *+Relation*)

Constrain variables to be a chain with respect to *Relation*. *Zs* is a list of finite domain variables that are a chain with respect to the partial order *Relation*, in the order they appear in the list. *Relation* must be #=, #=<, #>=, #< or #>. For example:

```
?- chain([X,Y,Z], #>=).
```

```
X#>=Y,
Y#>=Z.
```

fd_var(+*Var*)
True iff *Var* is a CLP(FD) variable.

fd_inf(+*Var*, -*Inf*)
Inf is the infimum of the current domain of *Var*.

fd_sup(+*Var*, -*Sup*)
Sup is the supremum of the current domain of *Var*.

fd_size(+*Var*, -*Size*)
Determine the size of a variable's domain. *Size* is the number of elements of the current domain of *Var*, or the atom **sup** if the domain is unbounded.

fd_dom(+*Var*, -*Dom*)
Dom is the current domain (see in/2) of *Var*. This predicate is useful if you want to reason about domains. It is not needed if you only want to display remaining domains; instead, separate your model from the search part and let the toplevel display this information via residual goals.