A.8 library(clpfd): Constraint Logic Programming over Finite Domains

author
Markus Triska

A.8.1 Introduction

This library provides CLP(FD): Constraint Logic Programming over Finite Domains.

CLP(FD) is an instance of the general CLP(.) scheme, extending logic programming with reasoning over specialised domains. CLP(FD) lets you reason about **integers**.

There are two major use cases of CLP(FD) constraints:

1. **declarative integer arithmetic**
2. solving **combinatorial problems** such as planning, scheduling and allocation tasks.

The predicates of this library can be classified as:

- *arithmetic* constraints like #=/2, #>/2 and #\=/2
- the *membership* constraints in/2 and ins/2
- *combinatorial* constraints like all_distinct/1
- *reification* and *reflection* predicates such as #<==>/2
- the *enumeration* predicates indomain/1, label/1 and labeling/2.

In most cases, arithmetic constraints are the only predicates you will ever need from this library. When reasoning over integers, simply replace low-level arithmetic predicates like is/2 and >/2 by the corresponding CLP(FD) constraints like #=/2 and #>/2 to honor and preserve declarative properties of your programs. For satisfactory performance, arithmetic constraints are implicitly rewritten at compilation time so that low-level fallback predicates are automatically used whenever possible.

Almost all Prolog programs also reason about integers. Therefore, it is highly advisable that you make CLP(FD) constraints available in all your programs. One way to do this is to put the following directive in your `~/.swiplrc` initialisation file:

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

All example programs that appear in the CLP(FD) documentation assume that you have done this.

Important concepts and principles of this library are illustrated by means of usage examples that are available in a public git repository: **github.com/triska/clpfd**

If you are used to the complicated operational considerations that low-level arithmetic primitives necessitate, then moving to CLP(FD) constraints may, due to their power and convenience, at first feel to you excessive and almost like cheating. It *isn't*. Constraints are an integral part of all popular Prolog systems, and they are designed to help you eliminate and avoid the use of low-level and less general primitives by providing declarative alternatives that are meant to be used instead.

When teaching Prolog, CLP(FD) constraints should be introduced *before* explaining low-level arithmetic predicates and their procedural idiosyncrasies. This is because constraints are easy to explain, understand and use due to their purely relational nature. In contrast, the modedness and directionality of low-level arithmetic primitives are impure limitations that are better deferred to more advanced lectures.

We recommend the following reference (PDF: metalevel.at/swiclpfd.pdf) for citing this library in scientific publications:

```
@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}
}
```

More information about CLP(FD) constraints and their implementation is contained in: **metalevel.at/drt.pdf**

The best way to discuss applying, improving and extending CLP(FD) constraints is to use the dedicated `clpfd` tag on stackoverflow.com. Several of the world's foremost CLP(FD) experts regularly participate in these discussions and will help you for free on this platform.

A.8.2 Arithmetic constraints

In modern Prolog systems, **arithmetic constraints** subsume and supersede low-level predicates over integers. The main advantage of arithmetic constraints is that they are true *relations* and can be used in all directions. For most programs, arithmetic constraints are the only predicates you will ever need from this library.

The arithmetic constraints are:

Expr1 `#=` Expr2	Expr1 equals Expr2
Expr1 `#\=` Expr2	Expr1 is not equal to Expr2
Expr1 `#>=` Expr2	Expr1 is greater than or equal to Expr2
Expr1 `#=<` Expr2	Expr1 is less than or equal to Expr2
Expr1 `#>` Expr2	Expr1 is greater than Expr2
Expr1 `#<` Expr2	Expr1 is less than Expr2

Expr1 and *Expr2* denote **arithmetic expressions**, which are:

integer	Given value
variable	Unknown integer
?(*variable*)	Unknown integer
-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

where *Expr* again denotes an arithmetic expression.

A.8.3 Declarative integer arithmetic

The arithmetic constraints #=/2, #>/2 etc. are meant to be used *instead* of the primitives is/2, =:=/2, >/2 etc. over integers. Throughout the following, it is assumed that you have put the following directive in your `~/.swiplrc` initialisation file to make CLP(FD) constraints available in all your programs:

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

An important advantage of arithmetic constraints is their purely relational nature. They are therefore easy to explain and use, and well suited for beginners and experienced Prolog programmers alike.

Consider for example the query:

```
?- X #> 3, X #= 5 + 2.
X = 7.
```

In contrast, when using low-level integer arithmetic, we get:

```
?- X > 3, X is 5 + 2.
ERROR: >/2: Arguments are not sufficiently instantiated
```

Due to the necessary operational considerations, the use of these low-level arithmetic predicates is considerably harder to understand and should therefore be deferred to more advanced lectures.

For supported expressions, CLP(FD) constraints are drop-in replacements of these low-level arithmetic predicates, often yielding more general programs.

Here is an example, relating each natural number to its factorial:

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

This relation 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 automatically rewrite arithmetic constraints at compilation time. The expansion's aim is to bring the performance of arithmetic constraints close to that of low-level arithmetic predicates whenever possible. To disable the expansion, set the flag `clpfd_goal_expansion` to `false`.

A.8.4 Combinatorial constraints

In addition to subsuming and replacing low-level arithmetic predicates, CLP(FD) constraints are often used to solve combinatorial problems such as planning, scheduling and allocation tasks. Among the most frequently used **combinatorial constraints** are all_distinct/1, global_cardinality/2 and cumulative/2. This library also provides several other constraints like disjoint2/1 and automaton/8, which are useful in more specialized applications.

A.8.5 Domains

Each CLP(FD) variable has an associated set of admissible integers, which we call the variable's **domain**. Initially, the domain of each CLP(FD) variable is the set of *all* integers. CLP(FD) constraints like #=/2, #>/2 and #\=/2 can at most reduce, and never extend, the domains of their arguments. The constraints in/2 and ins/2 let you explicitly state domains of CLP(FD) variables. The process of determining and adjusting domains of variables is called constraint **propagation**, and it is performed automatically by this library. When the domain of a variable contains only one element, then the variable is automatically unified to that element.

Domains are taken into account when further constraints are stated, and by enumeration predicates like labeling/2.

A.8.6 Residual goals

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

```
?- 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.

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

The answers emitted by the toplevel are called *residual programs*, and the goals that comprise each answer are called **residual goals**. In each case, and as for all pure programs, the residual program is declaratively equivalent to the original query. From the residual goals, it is clear that the constraint solver has deduced additional domain restrictions in many cases.

To inspect residual goals, it is best to let the toplevel display them for you. Wrap the call of your predicate into call_residue_vars/2 to make sure that all constrained variables are displayed. To make the constraints a variable is involved in available as a Prolog term for further reasoning within your program, use copy_term/3. For example:

```
?- X #= Y + Z, X in 0..5, copy_term([X,Y,Z], [X,Y,Z], Gs).
Gs = [clpfd: (X in 0..5), clpfd: (Y+Z#=X)],
X in 0..5,
Y+Z#=X.
```

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.

A.8.7 Core relations and search

Using CLP(FD) constraints to solve combinatorial tasks typically consists of two phases:

1. First, all relevant constraints are stated.
2. Second, if the domain of each involved variable is *finite*, then *enumeration*

predicates can be used to search for concrete solutions.

It is good practice to keep the modeling part, via a dedicated predicate called the **core relation**, separate from the actual search for solutions. This lets you observe termination and determinism properties of the core relation in isolation from the search, and more easily try different search strategies.

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:

```
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.
```

Notice that we are *not* using labeling/2 in this predicate, so that we can first execute and observe the modeling part in isolation. 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]),
91*A2+A4+10*B3#=90*A3+C5,
A3 in 5..8,
A4 in 2..8,
B3 in 2..8,
C5 in 2..8.
```

From this answer, we see that this core relation *terminates* and is in fact *deterministic*. Moreover, we see from the residual goals that the constraint solver has deduced more stringent bounds for all variables. Such observations are only possible if modeling and search parts are cleanly separated.

Labeling can then be used to search for solutions in a separate predicate or goal:

```
?- 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.

A.8.8 Optimisation

You can use labeling/2 to minimize or maximize the value of a CLP(FD) expression, and generate solutions in increasing or decreasing order of the value. See the labeling options `min(Expr)` and `max(Expr)`, respectively.

Again, to easily try different labeling options in connection with optimisation, we recommend to introduce a dedicated predicate for posting constraints, and to use `labeling/2` in a separate goal. This way, you can observe properties of the core relation in isolation, and try different labeling options without recompiling your code.

If necessary, you can use `once/1` to commit to the first optimal solution. However, it is often very valuable to see alternative solutions that are *also* optimal, so that you can choose among optimal solutions by other criteria. For the sake of purity and completeness, we recommend to avoid `once/1` and other constructs that lead to impurities in CLP(FD) programs.

A.8.9 Reification

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 either P or Q, but not both
`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.

When reasoning over Boolean variables, also consider using `library(clpb)` and its dedicated CLP(B) constraints.

A.8.10 Enabling monotonic CLP(FD)

In the default execution mode, CLP(FD) constraints still exhibit some non-relational properties. For example, *adding* constraints can yield new solutions:

```
?-          X #= 2, X = 1+1.
false.

?- X = 1+1, X #= 2, X = 1+1.
X = 1+1.
```

This behaviour is highly problematic from a logical point of view, and it may render declarative debugging techniques inapplicable.

Set the Prolog flag `clpfd_monotonic` to `true` to make CLP(FD) **monotonic**: This means that *adding* new constraints *cannot* yield new solutions. When this flag is `true`, you must wrap variables that occur in arithmetic expressions with the functor `(?)/1` or `(#)/1`. For example:

```
?- set_prolog_flag(clpfd_monotonic, true).
true.

?- #(X) #= #(Y) + #(Z).
#(Y)+ #(Z)#= #(X).

?-          X #= 2, X = 1+1.
ERROR: Arguments are not sufficiently instantiated
```

The wrapper can be omitted for variables that are already constrained to integers.

A.8.11 Custom constraints

You can 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:

```
:- 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.
```

A.8.12 Example: Eight queens puzzle

We illustrate the most important concepts of this library by means of the so-called *eight queens puzzle*. The task is to place 8 queens on an 8x8 chessboard such that none of the queens is under attack. This means that no two queens share the same row, column or diagonal.

To express this puzzle via CLP(FD) constraints, we must first pick a suitable representation. Since CLP(FD) constraints reason over *integers*, we must find a way to map the positions of queens to integers. Several such mappings are conceivable, and it is not immediately obvious which we should use. For this reason, *modeling* combinatorial problems via CLP(FD) constraints often necessitates some creativity and has been described as more of an art than a science.

In our concrete case, we observe that there must be exactly one queen per column. The following representation therefore suggests itself: We are looking for 8 integers, one for each column, where each integer denotes the *row* of the queen that is placed in the respective column, and which are subject to certain constraints.

In fact, let us now generalize the task to the so-called *N queens puzzle*, which is obtained by replacing 8 by *N* everywhere it occurs in the above description. We implement the above considerations in the **core relation** `n_queens/2`, where the first argument is the number of queens (which is identical to the number of rows and columns of the generalized chessboard), and the second argument is a list of *N* integers that represents a solution in the form described above.

```
n_queens(N, Qs) :-
        length(Qs, N),
        Qs ins 1..N,
        safe_queens(Qs).

safe_queens([]).
```

```
safe_queens([Q|Qs]) :- safe_queens(Qs, Q, 1), safe_queens(Qs).

safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-
        Q0 #\= Q,
        abs(Q0 - Q) #\= D0,
        D1 #= D0 + 1,
        safe_queens(Qs, Q0, D1).
```

Note that all these predicates can be used in *all directions*: You can use them to *find* solutions, *test* solutions and *complete* partially instantiated solutions.

The original task can be readily solved with the following query:

```
?- n_queens(8, Qs), label(Qs).
Qs = [1, 5, 8, 6, 3, 7, 2, 4] .
```

Using suitable labeling strategies, we can easily find solutions with 80 queens and more:

```
?- n_queens(80, Qs), labeling([ff], Qs).
Qs = [1, 3, 5, 44, 42, 4, 50, 7, 68|...] .

?- time((n_queens(90, Qs), labeling([ff], Qs))).
% 5,904,401 inferences, 0.722 CPU in 0.737 seconds (98% CPU, 8183406 Lips)
Qs = [1, 3, 5, 50, 42, 4, 49, 7, 59|...] .
```

Experimenting with different search strategies is easy because we have separated the core relation from the actual search.

A.8.13 Applications

CLP(FD) applications that we find particularly impressive and worth studying include:

- Michael Hendricks uses CLP(FD) constraints for flexible reasoning about *dates* and *times* in the `julian` package.
- Julien Cumin uses CLP(FD) constraints for integer arithmetic in `Brachylog`.

A.8.14 Acknowledgments

This library gives you a glimpse of what **SICStus Prolog** can do. The API is intentionally mostly compatible with that of SICStus Prolog, so that you can easily switch to a much more feature-rich and much faster CLP(FD) system when you need it. I thank Mats Carlsson, the designer and main implementor of SICStus Prolog, for his elegant example. I first encountered his system as part of the excellent **GUPU** teaching environment by Ulrich Neumerkel. Ulrich was also the first and most determined tester of the present system, filing hundreds of comments and suggestions for improvement. Tom Schrijvers has contributed several constraint libraries to SWI-Prolog, and I learned a lot from his coding style and implementation examples. Bart Demoen was a driving force behind the implementation of attributed variables in SWI-Prolog, and this library could not even have started without his prior work and contributions. Thank you all!

A.8.15 CLP(FD) predicate index

In the following, each CLP(FD) predicate is described in more detail.

We recommend the following link to refer to this manual:

http://eu.swi-prolog.org/man/clpfd.html

?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*. See in/2 for the syntax 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)`. See labeling/2.

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. See core relations and search for usage advice.

all_different(*+Vars*)

Like all_distinct/1, but with weaker propagation.

all_distinct(*+Vars*)

True iff *Vars* are pairwise distinct. For example, all_distinct/1 can detect that not all variables can assume distinct values given the following domains:

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

True iff the scalar product of *Cs* and *Vs* is in relation *Rel* to *Expr*. *Cs* is a list of integers, *Vs* is a list of variables and integers. *Rel* is #=, #\=, #<, #>, #=< or #>=.

?X #>= ?Y

Same as *Y #=< X*. When reasoning over integers, replace >=/2 by #>=/2 to obtain more general relations. See declarative integer arithmetic.

?X #=< ?Y

The arithmetic expression *X* is less than or equal to *Y*. When reasoning over integers, replace =</2 by #=</2 to obtain more general relations. See declarative integer

arithmetic.

?X #= ?Y

The arithmetic expression *X* equals *Y*. When reasoning over integers, replace is/2 by #=/2 to obtain more general relations. See declarative integer arithmetic.

?X #\= ?Y

The arithmetic expressions *X* and *Y* evaluate to distinct integers. When reasoning over integers, replace =\=/2 by #\=/2 to obtain more general relations. See declarative integer arithmetic.

?X #> ?Y

Same as *Y #< X*. When reasoning over integers, replace >/2 by #>/2 to obtain more general relations See declarative integer arithmetic.

?X #< ?Y

The arithmetic expression *X* is less than *Y*. When reasoning over integers, replace </2 by #</2 to obtain more general relations. See declarative integer arithmetic.

In addition to its regular use in tasks 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

Q does *not* hold. See reification.

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. See reification.

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:

```
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*. See reification.

?P #<== ?Q

Q implies *P*. See reification.

?P #/\ ?Q

P and *Q* hold. See reification.

?P #\/ ?Q

P or *Q* holds. See reification.

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.
```

?P #\ ?Q

Either *P* holds or *Q* holds, but not both. See reification.

lex_chain(*+Lists*)

Lists are lexicographically non-decreasing.

tuples_in(*+Tuples, +Relation*)

True iff all *Tuples* are elements of *Relation*. Each element of the list *Tuples* is a list of integers or finite domain variables. *Relation* is a list of lists of integers. 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.

```
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*)

Describes a set of non-overlapping tasks. *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, [])`. See global_cardinality/3.

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 iff 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)])`. See cumulative/2.

cumulative(*+Tasks, +Options*)

Schedule with a limited resource. *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 iff at each time slot 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. *Options* is a list of options. Currently, the only supported option is:

limit(*L*)
The integer *L* is the global resource limit. Default is 1.

For example, given the following predicate that relates three tasks of durations 2 and 3 to a list containing their starting times:

```
tasks_starts(Tasks, [S1,S2,S3]) :-
        Tasks = [task(S1,3,_,1,_),
                 task(S2,2,_,1,_),
                 task(S3,2,_,1,_)].
```

We can use cumulative/2 as follows, and obtain a schedule:

```
?- tasks_starts(Tasks, Starts), Starts ins 0..10,
   cumulative(Tasks, [limit(2)]), label(Starts).
Tasks = [task(0, 3, 3, 1, _G36), task(0, 2, 2, 1, _G45), ...],
Starts = [0, 0, 2] .
```

disjoint2(*+Rectangles*)

True iff *Rectangles* are not overlapping. *Rectangles* is a list of terms of the form F(X_i, W_i, Y_i, H_i), where F is any functor, and the arguments are finite domain variables or integers that denote, respectively, the X coordinate, width, Y coordinate and height of each rectangle.

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

Describes a list of finite domain variables with a finite automaton. Equivalent to `automaton(Vs, _, Vs, 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:

```
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)]).
```

Example query:

```
?- 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)

Describes a list of finite domain variables with a finite automaton. True iff 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* symbolically refer to previous counter values, and variables occurring in *Template* refer to the current element of *Sequence*. When a transition containing arithmetic expressions is taken, each counter is updated according to the result of the corresponding expression. When a transition without arithmetic expressions is taken, all counters remain unchanged. *Counters* is a list of variables. *Initials* is a list of finite domain variables or integers denoting, in the same order, the initial value of each counter. These values are related to *Finals* according to the arithmetic expressions of the taken transitions.

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:

```
sequence_inflexions(Vs, N) :-
        variables_signature(Vs, Sigs),
        automaton(Sigs, _, 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]].
```

This predicate is useful in many constraint programs. Consider for instance Sudoku:

```
sudoku(Rows) :-
```

```
        length(Rows, 9), maplist(same_length(Rows), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs), blocks(Ds, Es, Fs), blocks(Gs, Hs, Is).

blocks([], [], []).
blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
        all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
        blocks(Ns1, Ns2, Ns3).

problem(1, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).
```

Sample query:

```
?- problem(1, Rows), sudoku(Rows), maplist(writeln, Rows).
[9,8,7,6,5,4,3,2,1]
[2,4,6,1,7,3,9,8,5]
[3,5,1,9,2,8,7,4,6]
[1,2,8,5,3,7,6,9,4]
[6,3,4,8,9,2,1,5,7]
[7,9,5,4,6,1,8,3,2]
[5,1,9,2,8,6,4,7,3]
[4,7,2,3,1,9,5,6,8]
[8,6,3,7,4,5,2,1,9]
Rows = [[9, 8, 7, 6, 5, 4, 3, 2|...], ... , [...|...]].
```

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

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

This predicate allows you to make several predicates over integers deterministic while preserving their generality and completeness. For example:

```
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, because first argument indexing can distinguish the two different clauses:

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

The predicate can still be used in all directions, including the most general query:

```
?- n_factorial(N, F).
N = 0,
F = 1 ;
N = F, F = 1 ;
N = F, F = 2 .
```

chain(*+Zs, +Relation*)

Zs form 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*)

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.

For example, to implement a custom labeling strategy, you may need to inspect the current domain of a finite domain variable. With the following code, you can convert a *finite* domain to a list of integers:

```
dom_integers(D, Is) :- phrase(dom_integers_(D), Is).

dom_integers_(I)      --> { integer(I) }, [I].
dom_integers_(L..U)   --> { numlist(L, U, Is) }, Is.
dom_integers_(D1\/D2) --> dom_integers_(D1), dom_integers_(D2).
```

Example:

```
?- X in 1..5, X #\= 4, fd_dom(X, D), dom_integers(D, Is).
D = 1..3\/5,
Is = [1,2,3,5],
X in 1..3\/5.
```