CLP(FD):
Constraint Logic Programming over Finite Domains

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

1 Introduction

This document is a *shortened version* of the CLP(FD) library documentation that ships with SWI-Prolog. The full version is available at:

```
http://www.swi-prolog.org/man/clpfd.html
```

CLP(FD) is an instance of the general CLP(·) scheme, extending Prolog with reasoning over specialized domains. `library(clpfd)` provides Constraint Logic Programming over *Finite Domains*, which means sets of *integers*.

There are two major use cases of `library(clpfd)`:

1. CLP(FD) constraints provide **declarative integer arithmetic**: They implement pure *relations* between integer expressions and can be used in all directions, also if parts of expressions are variables.

2. In connection with enumeration predicates and more complex constraints, CLP(FD) is often used to model and solve **combinatorial problems** such as planning, scheduling and allocation tasks.

We *strongly recommend* that you use CLP(FD) constraints *instead* of lower-level arithmetic predicates over integers. 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 non-declarative limitations that make it much harder to reason about your programs.

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 many Prolog systems and are available to help you eliminate and avoid, as far as possible, the use of lower-level and less general primitives by providing declarative alternatives that are meant to be used instead.

For satisfactory performance, arithmetic constraints are implicitly rewritten at compilation time so that lower-level fallback predicates are automatically used whenever possible.

2 Arithmetic constraints

A finite domain *arithmetic expression* is one of:

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

Arithmetic *constraints* are relations between arithmetic expressions. The most important arithmetic constraints are:

$Expr_1$ `#>=` $Expr_2$	$Expr_1$ is greater than or equal to $Expr_2$
$Expr_1$ `#=<` $Expr_2$	$Expr_1$ is less than or equal to $Expr_2$
$Expr_1$ `#=` $Expr_2$	$Expr_1$ equals $Expr_2$
$Expr_1$ `#\=` $Expr_2$	$Expr_1$ is not equal to $Expr_2$
$Expr_1$ `#>` $Expr_2$	$Expr_1$ is greater than $Expr_2$
$Expr_1$ `#<` $Expr_2$	$Expr_1$ is less than $Expr_2$

3 Declarative integer arithmetic

CLP(FD) constraints let you declaratively express *integer arithmetic*. The CLP(FD) constraints `(#=)/2`, `(#>)/2` etc. are meant to be used instead of the corresponding primitives `(is)/2`, `(=:=)/2`, `(>)/2` etc. over integers.

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 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:

```
:- 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(38, F).
F = 523022617466601111760007224100074291200000000 ;
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 lower-level arithmetic predicates whenever possible. To disable the expansion, set the flag `clpfd_goal_expansion` to `false`.

4 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.

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. The constraints `(in)/2` and `(ins)/2` are the primary means to specify tighter domains of variables.

Here are example queries and the system's declaratively equivalent answers:

```
?- X in 100..sup.
X in 100..sup.

?- X in 1..5 \/ 3..12.
X in 1..12.

?- [X,Y,Z] ins 0..3.
X in 0..3,
Y in 0..3,
Z in 0..3.
```

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

6 Examples

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

```
?- use_module(library(clpfd)).
% library(clpfd) compiled into clpfd 0.06 sec
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.

?- 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.

7 Core relations and search

In addition to being declarative replacements for low-level arithmetic predicates, CLP(FD) constraints are also often used to solve combinatorial problems such as planning, scheduling and allocation tasks. To let you conveniently model and solve such problems, this library provides several constraints beyond typical integer arithmetic, such as `all_distinct/1`, `global_cardinality/2` and `cumulative/1`.

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:

```
:- 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.
```

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.

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.

9 Advanced topics

If you set the flag `clpfd_monotonic` to `true`, then CLP(FD) is monotonic: 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`. For example, ?(X) #= ?(Y) + ?(Z). The wrapper can be omitted for variables that are already constrained to integers.

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.

10 Enumeration predicates

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:

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 1..3, 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.

11 Global constraints

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

all_distinct(+*Ls*)
Like `all_different/1`, with stronger propagation.

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

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 #>=.

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.

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.

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, []).`

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.

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

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.

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`.

automaton(+*Seq, ?T, +Sigs, +Nodes, +Arcs, +Counters, +Is, ?Fs*)
A powerful global constraint, describing a list of finite domain variables with a finite automaton. Please see the full version of the manual for more information.

transpose(+*Matrix, ?Transpose*)
Transpose a list of lists of the same length.

zcompare(*?Order, ?A, ?B*)
Analogous to `compare/3`, with finite domain variables *A* and *B*.

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 #>.

12 Reflection predicates

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.