34 Constraint Logic Programming over Finite Domains

34.1 Introduction

The clp(FD) solver described in this chapter is an instance of the general Constraint Logic Programming scheme introduced in [Jaffar & Michaylov 87]. This constraint domain is particularly useful for modeling discrete optimization and verification problems such as scheduling, planning, packing, timetabling etc. The treatise [Van Hentenryck 89] is an excellent exposition of the theoretical and practical framework behind constraint solving in finite domains, and summarizes the work up to 1989.

This solver has the following highlights:

- Two classes of constraints are handled internally: primitive constraints and global constraints.
- The constraints described in this chapter are automatically translated to conjunctions of primitive and global library constraints.
- The truth value of a primitive constraint can be reflected into a 0/1-variable (reification).
- New primitive constraints can be added by writing so-called indexicals.
- New global constraints can be written in Prolog, by means of a programming interface.

This library fully supports multiple SICStus run-times in a process.

The rest of this chapter is organized as follows: How to load the solver and how to write simple programs is explained in Section 34.2 [CLPFD Interface], page 446. A description of all constraints that the solver provides is contained in Section 34.3 [Available Constraints], page 450. The predicates for searching for solution are documented in Section 34.4 [Enumeration Predicates], page 465. The predicates for getting execution statistics are documented in Section 34.5 [Statistics Predicates], page 469. A few example programs are given in Section 34.10 [Example Programs], page 487. Finally, Section 34.11 [Syntax Summary], page 490 contains syntax rules for all expressions.

The following sections discuss advanced features and are probably only relevant to experienced users: How to control the amount of information presented in answers to queries is explained in Section 34.6 [Answer Constraints], page 470. The solver’s execution mechanism and primitives are described in Section 34.7 [The Constraint System], page 470. How to add new global constraints via a programming interface is described in Section 34.8 [Defining Global Constraints], page 471. How to define new primitive constraints with indexicals is described in Section 34.9 [Defining Primitive Constraints], page 479.
34.1.1 Referencing this Software

When referring to this implementation of clp(FD) in publications, please use the following reference:


34.1.2 Acknowledgments

The first version of this solver was written as part of Key Hyckenberg’s MSc thesis in 1995, with contributions from Greger Ottosson at the Computing Science Department, Uppsala University. The code was later rewritten by Mats Carlsson. Péter Széredi contributed material for this manual chapter.

The development of this software was supported by the Swedish National Board for Technical and Industrial Development (NUTEK) under the auspices of Advanced Software Technology (ASTEC) Center of Competence at Uppsala University.

We include a collection of examples, some of which have been distributed with the INRIA implementation of clp(FD) [Diaz & Codognet 93].

34.2 Solver Interface

The solver is available as a library module and can be loaded with a query

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

The solver contains predicates for checking the consistency and entailment of finite domain constraints, as well as solving for solution values for your problem variables.

In the context of this constraint solver, a finite domain is a subset of small integers, and a finite domain constraint denotes a relation over a tuple of small integers. Hence, only small integers and unbound variables are allowed in finite domain constraints.

All domain variables, i.e. variables that occur as arguments to finite domain constraints get associated with a finite domain, either explicitly declared by the program, or implicitly imposed by the constraint solver. Temporarily, the domain of a variable may actually be infinite, if it does not have a finite lower or upper bound. If during the computation a variable receives a new lower or upper bound that cannot be represented as a small integer, an overflow condition is issued. This is expressed as silent failure or as a representation error, subject to the overflow option of fd_flag/3.
The domain of all variables gets narrower and narrower as more constraints are added. If a domain becomes empty, the accumulated constraints are unsatisfiable, and the current computation branch fails. At the end of a successful computation, all domains have usually become singletons, i.e. the domain variables have become assigned.

The domains do not become singletons automatically. Usually, it takes some amount of search to find an assignment that satisfies all constraints. It is the programmer’s responsibility to do so. If some domain variables are left unassigned in a computation, the garbage collector will preserve all constraint data that is attached to them.

**Please note:** the behavior of the predicates `assert/1`, `findall/3`, `raise_exception/1` and friends is undefined on non-ground terms containing domain variables. If you need to apply these operations to a term containing domain variables, `fd_copy_term/3` may be used to decompose the term into a template and a symbolic representation of the relevant constraints.

The heart of the constraint solver is a scheduler for indexicals [Van Hentenryck et al. 92] and global constraints. Both entities act as coroutines performing incremental constraint solving or entailment checking. They wake up by changes in the domains of its arguments. All constraints provided by this package are implemented as indexicals or global constraints. New constraints can be defined by the user.

Indexicals are reactive functional rules, which take part in the solver’s basic constraint solving algorithm, whereas each global constraint is associated with its particular constraint solving algorithm. The solver maintains two scheduling queues, giving priority to the queue of indexicals.

The feasibility of integrating the indexical approach with a Prolog based on the WAM was clearly demonstrated by Diaz’s clp(FD) implementation [Diaz & Codognet 93], one of the fastest finite domains solvers around.

### 34.2.1 Posting Constraints

A constraint is called as any other Prolog predicate. When called, the constraint is posted to the store. For example:

```
| ?- X in 1..5, Y in 2..8, X+Y #= T.
X in 1..5,
Y in 2..8,
T in 3..13

| ?- X in 1..5, T in 3..13, X+Y #= T.
X in 1..5,
T in 3..13,
Y in -2..12
```
Note that the answer constraint shows the domains of nonground query variables, but not any constraints that may be attached to them.

34.2.2 A Constraint Satisfaction Problem

Constraint satisfaction problems (CSPs) are a major class of problems for which this solver is ideally suited. In a CSP, the goal is to pick values from pre-defined domains for certain variables so that the given constraints on the variables are all satisfied.

As a simple CSP example, let us consider the Send More Money puzzle. In this problem, the variables are the letters S, E, N, D, M, O, R, and Y. Each letter represents a digit between 0 and 9. The problem is to assign a value to each digit, such that SEND + MORE equals MONEY.

A program that solves the puzzle is given below. The program contains the typical three steps of a clp(FD) program:

1. declare the domains of the variables
2. post the problem constraints
3. look for a feasible solution via backtrack search, or look for an optimal solution via branch-and-bound search

Sometimes, an extra step precedes the search for a solution: the posting of surrogate constraints to break symmetries or to otherwise help prune the search space. No surrogate constraints are used in this example.

The domains of this puzzle are stated via the domain/3 goal and by requiring that S and M be greater than zero. The two problem constraint of this puzzle are the equation (sum/8) and the constraint that all letters take distinct values (all_different/1). Finally, the backtrack search is performed by labeling/2. Different search strategies can be encoded in the Type parameter. In the example query, the default search strategy is used (select the leftmost variable, try values in ascending order).
### 34.2.3 Reified Constraints

Instead of merely posting constraints it is often useful to reflect its truth value into a 0/1-variable $B$, so that:

- the constraint is posted if $B$ is set to 1
- the negation of the constraint is posted if $B$ is set to 0
- $B$ is set to 1 if the constraint becomes entailed
- $B$ is set to 0 if the constraint becomes disentailed

This mechanism is known as reification. Several frequently used operations can be defined in terms of reified constraints, such as blocking implication [Saraswat 90] and the cardinality operator [Van Hentenryck & Deville 91], to name a few. A reified constraint is written:

```
| ?- Constraint #<=> B.
```

where $Constraint$ is reifiable. As an example of a constraint that uses reification, consider $\text{exactly}(X, L, N)$, which is true if $X$ occurs exactly $N$ times in the list $L$. It can be defined thus:
exactly(_, [], 0).
exactly(X, [Y|L], N) :-
    X #= Y #<=> B,
    N #= M+B,
    exactly(X, L, M).

34.3 Available Constraints

This section describes the classes of constraints that can be used with this solver.

34.3.1 Arithmetic Constraints

\(?Expr \ RelOp \ ?Expr\)

defines an arithmetic constraint. The syntax for \(Expr\) and \(RelOp\) is defined by a grammar (see Section 34.11.2 [Syntax of Arithmetic Expressions], page 492). Note that the expressions are not restricted to being linear. Constraints over non-linear expressions, however, will usually yield less constraint propagation than constraints over linear expressions.

Arithmetic constraints can be reified as e.g.:

\[- ?- X \text{ in } 1..2, Y \text{ in } 3..5, X \#=< Y \#<=> B.\]

\[B = 1,\]

\[X \text{ in } 1..2,\]

\[Y \text{ in } 3..5\]

Linear arithmetic constraints, except equalities, maintain interval-consistency and their reified versions detect interval-entailment and -disentailment; see Section 34.7 [The Constraint System], page 470.

The following constraints are among the library constraints that general arithmetic constraints compile to. They express a relation between a sum or a scalar product and a value, using a dedicated algorithm, which avoids creating any temporary variables holding intermediate values. If you are computing a sum or a scalar product, it can be much more efficient to compute lists of coefficients and variables and post a single sum or scalar product constraint than to post a sequence of elementary constraints.

\[\text{sum}(+Xs, +RelOp, ?Value)\]

where \(Xs\) is a list of integers or domain variables, \(RelOp\) is a relational symbol as above, and \(Value\) is an integer or a domain variable. True if \(Xs \ RelOp \ Value\). Cannot be reified.

\[\text{scalar} \_ \text{product}(+Coeffs, +Xs, +RelOp, ?Value)\]

where \(Coeffs\) is a list of length \(n\) of integers, \(Xs\) is a list of length \(n\) of integers or domain variables, \(RelOp\) is a relational symbol as above, and \(Value\) is an integer or a domain variable. True if \(Coeffs*Xs \ RelOp \ Value\). Cannot be reified.
The following constraint is a special case of `scalar_product/4` where `RelOp` is `#=`, and which is domain-consistent in the `Xs`:

\[ \text{knapsack}(\text{Coeffs}, Xs, ?Value) \]

where `Coeffs` is a list of length `n` of non-negative integers, `Xs` is a list of length `n` of non-negative integers or domain variables, and `Value` is an integer or a domain variable. Any domain variables must have finite bounds. True if `Coeffs*Xs = Value`. Cannot be reified.

### 34.3.2 Membership Constraints

\[ \text{domain}(\text{Variables}, \text{Min}, \text{Max}) \]

where `Variables` is a list of domain variables or integers, `Min` is an integer or the atom `inf` (minus infinity), and `Max` is an integer or the atom `sup` (plus infinity). True if the variables all are elements of the range `Min .. Max`. Cannot be reified.

\[ ?X \text{ in } \text{Range} \]

defines a membership constraint. `X` is an integer or a domain variable and `Range` is a `ConstantRange` (see Section 34.11.1 [Syntax of Indexicals], page 491). True if `X` is an element of the range.

\[ ?X \text{ in_set } \text{FDSet} \]

defines a membership constraint. `X` is an integer or a domain variable and `FDSet` is an FD set term (see Section 34.8.3 [FD Set Operations], page 475). True if `X` is an element of the FD set.

`in/2` and `in_set/2` constraints can be reified. They maintain domain-consistency and their reified versions detect domain-entailment and -disentailment; see Section 34.7 [The Constraint System], page 470.

### 34.3.3 Propositional Constraints

Propositional combinators can be used to combine reifiable constraints into propositional formulae over such constraints. Such formulae are goal expanded by the system into sequences of reified constraints and arithmetic constraints. For example,

\[ X \ #= \ 4 \ 
\text{\ or } \ Y \ #= \ 6 \]

expresses the disjunction of two equality constraints.

The leaves of propositional formulae can be reifiable constraints, the constants 0 and 1, or 0/1-variables. New primitive, reifiable constraints can be defined with indexicals as described in Section 34.9 [Defining Primitive Constraints], page 479. The following propositional combinators are available:

\[ #\setminus :Q \]
True if the constraint $Q$ is false.

$P \#\setminus :Q$

True if the constraints $P$ and $Q$ are both true.

$P \#\setminus :Q$

True if exactly one of the constraints $P$ and $Q$ is true.

$P \#\set\setminus :Q$

True if at least one of the constraints $P$ and $Q$ is true.

$P \#=> :Q$

$Q \#<=: P$

True if the constraint $Q$ is true or the constraint $P$ is false.

$P \#<=:Q$

True if the constraints $P$ and $Q$ are both true or both false.

Note that the reification scheme introduced in Section 34.2.3 [Reified Constraints], page 449 is a special case of a propositional constraint.

### 34.3.4 Combinatorial Constraints

The constraints listed here are sometimes called symbolic constraints. They are currently not reifiable. Unless documented otherwise, they maintain (at most) interval-consistency in their arguments; see Section 34.7 [The Constraint System], page 470.

\textbf{count(+Val,+List,+RelOp,+Count)}

where $Val$ is an integer, $List$ is a list of integers or domain variables, $Count$ an integer or a domain variable, and $RelOp$ is a relational symbol as in Section 34.3.1 [Arithmetic Constraints], page 450. True if $N$ is the number of elements of $List$ that are equal to $Val$ and $N \ RelOp \ Count$. Thus, $\text{count}/4$ is a generalization of $\text{exactly}/3$ (not an exported constraint), which was used in an example earlier.

$\text{count}/4$ maintains domain-consistency, but in practice, the following constraint is a better alternative.

\textbf{global_cardinality(+Xs,+Vals)}

\textbf{global_cardinality(+Xs,+Vals,+Options)}

where $Xs = [X_1, \ldots, X_d]$ is a list of integers or domain variables, and $Vals = [K_1 - V_1, \ldots, K_n - V_n]$ is a list of pairs where each key $K_i$ is a unique integer and $V_i$ is a domain variable or an integer. True if every element of $Xs$ is equal to some key and for each pair $K_i - V_i$, exactly $V_i$ elements of $Xs$ are equal to $K_i$.

If either $Xs$ or $Vals$ is ground, and in many other special cases, $\text{global_cardinality}/[2,3]$ maintains domain-consistency, but generally, interval-consistency cannot be guaranteed. A domain-consistency algorithm [Regin 96] is used, roughly linear in the total size of the domains.
Options is a list of zero or more of the following:

\texttt{cost(Cost, Matrix)}

A cost is associated with the constraint and reflected into the domain variable \texttt{Cost}. \texttt{Matrix} should be a $d \times n$ matrix, represented as a list of $d$ lists, each of length $n$. Assume that each $X_i$ equals $K_{p_i}$. The cost of the constraint is then $\text{Matrix}[1, p_1] + \cdots + \text{Matrix}[d, p_d]$. With this option, a domain-consistency algorithm [Regin 99] is used, the complexity of which is roughly $O(d(m + n \log n))$ where $m$ is the total size of the domains.

\texttt{element(?X, +List, ?Y)}

where $X$ and $Y$ are integers or domain variables and \texttt{List} is a list of integers or domain variables. True if the $X$:th element of \texttt{List} is $Y$. Operationally, the domains of $X$ and $Y$ are constrained so that for every element in the domain of $X$, there is a compatible element in the domain of $Y$, and vice versa.

This constraint uses an optimized algorithm for the special case where \texttt{List} is ground.

\texttt{element/3} maintains domain-consistency in $X$ and interval-consistency in \texttt{List} and $Y$.

\texttt{relation(?X, +MapList, ?Y)}

where $X$ and $Y$ are integers or domain variables and \texttt{MapList} is a list of \texttt{integer-ConstantRange} pairs, where the integer keys occur uniquely (see Section 34.11.1 [Syntax of Indexicals], page 491). True if \texttt{MapList} contains a pair $X$-$R$ and $Y$ is in the range denoted by $R$.

Operationally, the domains of $X$ and $Y$ are constrained so that for every element in the domain of $X$, there is a compatible element in the domain of $Y$, and vice versa.

If \texttt{MapList} is not ground, the constraint must be wrapped in \texttt{call/1} to postpone goal expansion until runtime.

An arbitrary binary constraint can be defined with \texttt{relation/3}. \texttt{relation/3} is implemented in terms of the following, more general constraint, with which arbitrary relations can be defined compactly:

\texttt{case(+Template, +Tuples, +Dag)}

\texttt{case(+Template, +Tuples, +Dag, +Options)}

\texttt{Template} is an arbitrary non-ground Prolog term. Its variables are merely place-holders; they should not occur outside the constraint nor inside \texttt{Tuples}.

\texttt{Tuples} is a list of terms of the same shape as \texttt{Template}. They should not share any variables with \texttt{Template}.

\texttt{Dag} is a list of \texttt{nodes} of the form \texttt{node(ID, X, Successors)}, where $X$ is a place-holder variable. The set of all $X$ should equal the set of variables in \texttt{Template}. The first node in the list is the \texttt{root node}. Let \texttt{rootID} denote its ID.

Nodes are either \texttt{internal nodes} or \texttt{leaf nodes}. In the former case, \texttt{Successors} is a list of terms \texttt{(Min..Max)-ID2}, where \texttt{ID2} refers to a child node. In the latter case, \texttt{Successors} is a list of terms \texttt{(Min..Max)}. In both cases, the \texttt{(Min..Max)} should form disjoint intervals.
ID is a unique, integer identifier of a node.
Each path from the root node to a leaf node corresponds to one set of tuples admitted by the relation expressed by the constraint. Each variable in Template should occur exactly once on each path, and there must not be any cycles.

Options is a list of zero or more of the following. It can be used to control the waking and pruning conditions of the constraint, as well as to identify the leaf nodes reached by the tuples:

leaves(TLeaf,Leaves)

TLeaf is a place-holder variable. Leaves is a list of variables of the same length as Tuples. This option effectively extends the relation by one argument, corresponding to the ID of the leaf node reached by a particular tuple.

on(Spec) Specifies how eagerly the constraint should react to domain changes of X.

prune(Spec)

Specifies the extent to which the constraint should prune the domain of X.

Spec is one of the following, where X is a place-holder variable occurring in Template or equal to TLeaf:

dom(X) wake up when the domain of X has changed, resp. perform full pruning on X. This is the default for all variables mentioned in the constraint.

min(X) wake up when the lower bound of X has changed, resp. prune only the lower bound of X.

max(X) wake up when the upper bound of X has changed, resp. prune only the upper bound of X.

minmax(X) wake up when the lower or upper bound of X has changed, resp. prune only the bounds of X.

val(X) wake up when X has become ground, resp. only prune X when its domain has been narrowed to a singleton.

none(X) ignore domain changes of X, resp. never prune X.

The constraint holds if path(rootID,Tuple,Leaf) holds for each Tuple in Tuples and Leaf is the corresponding element of Leaves if given (otherwise, Leaf is a free variable).

path(ID,Tuple,Leaf) holds if Dag contains a term node(ID,Var,Successors), Var is the unique kth element of Template, i is the kth element of Tuple, and:

• The node is an internal node, and
  1. Successors contains a term (Min..Max)-Child,
  2. Min \leq i \leq Max, and
  3. path(Child,Tuple,Leaf) holds; or
The node is a leaf node, and

1. **Successors** contains a term \((\text{Min}..\text{Max})\),
2. \(\text{Min} \leq i \leq \text{Max}\), and \(\text{Leaf} = \text{ID}\).

For example, recall that \(\text{element}(X,L,Y)\) wakes up when the domain of \(X\) or the lower or upper bound of \(Y\) has changed, performs full pruning of \(X\), but only prunes the bounds of \(Y\). The following two constraints:

\[
\text{element}(X, [1,1,1,1,2,2,2,2], Y),
\text{element}(X, [10,10,20,20,10,10,30,30], Z)
\]

can be replaced by the following single constraint, which is equivalent declaratively as well as wrt. pruning and waking. The fourth argument illustrates the leaf feature:

\[
\text{elts}(X, Y, Z, L) :-
\text{case}(f(A,B,C), [f(X,Y,Z)],
    [\text{node}(0, A, [(1..2)-1,(3..4)-2,(5..6)-3,(7..8)-4]),
     \text{node}(1, B, [(1..1)-5]),
     \text{node}(2, B, [(1..1)-6]),
     \text{node}(3, B, [(2..2)-5]),
     \text{node}(4, B, [(2..2)-7]),
     \text{node}(5, C, [(10..10)]),
     \text{node}(6, C, [(20..20)]),
     \text{node}(7, C, [(30..30)]),
    [\text{on}(\text{dom}(A)), \text{on}(\text{minmax}(B)), \text{on}(\text{minmax}(C)),
     \text{prune}(\text{dom}(A)), \text{prune}(\text{minmax}(B)), \text{prune}(\text{minmax}(C)),
     \text{leaves}(\_, [L])]).
\]

The DAG of the previous example has the following shape:
DAG corresponding to `elts/4`

A couple of sample queries:
| ?- elts(X, Y, Z, L).
L in 5..7,
X in 1..8,
Y in 1..2,
Z in 10..30

| ?- elts(X, Y, Z, L), Z #>= 15.
L in 6..7,
X in(3..4)\/(7..8),
Y in 1..2,
Z in 20..30

| ?- elts(X, Y, Z, L), Y = 1.
Y = 1,
L in 5..6,
X in 1..4,
Z in 10..20

| ?- elts(X, Y, Z, L), L = 5.
Z = 10,
X in(1..2)\/(5..6),
Y in 1..2

all_different(+Variables)
all_different(+Variables, +Options)
all_distinct(+Variables)
all_distinct(+Variables, +Options)

where Variables is a list of domain variables with bounded domains or integers. Each variable is constrained to take a value that is unique among the variables. Declaratively, this is equivalent to an inequality constraint for each pair of variables.

Options is a list of zero or more of the following:

on(On) How eagerly to wake up the constraint. One of:

- dom (the default for all_distinct/[1,2] and assignment/[2,3]), to wake up when the domain of a variable is changed;
- min to wake up when the lower bound of a domain is changed;
- max to wake up when the upper bound of a domain is changed;
- minmax to wake up when some bound of a domain is changed;
- val (the default for all_different/[1,2]), to wake up when a variable becomes ground.
consistency(Cons)
Which algorithm to use, one of:

**global**  The default for all_distinct/[1,2] and assignment/[2,3]. A domain-consistency algorithm [Regin 94] is used, roughly linear in the total size of the domains.

**local**  The default for all_different/[1,2]. An algorithm achieving exactly the same pruning as a set of pairwise inequality constraints is used, roughly linear in the number of variables.

**bound**  An interval-consistency algorithm [Mehlhorn 00] is used. This algorithm is nearly linear in the number of variables and values.

The following is a constraint over two lists of length n of variables. Each variable is constrained to take a value in [1, n] that is unique for its list. Furthermore, the lists are dual in a sense described below.

assignment(+Xs, +Ys)
assignment(+Xs, +Ys, +Options)
where Xs = [X₁, ..., Xₙ] and Ys = [Y₁, ..., Yₙ] are lists of domain variables or integers. True if all Xᵢ, Yᵢ ∈ [1, n] and Xᵢ = j ≡ Yᵢ = i.

Options is a list of zero or more of the following, where Boolean must be true or false (false is the default):

**on(On)**  Same meaning as for all_different/2.

**consistency(Cons)**  Same meaning as for all_different/2.

**circuit(Boolean)**  If true, circuit(Xs, Ys) must hold for the constraint to be true.

**cost(Cost, Matrix)**  A cost is associated with the constraint and reflected into the domain variable Cost. Matrix should be an n × n matrix, represented as a list of lists. The cost of the constraint is Matrix[1, X₁] + · · · + Matrix[n, Xₙ].

With this option, a domain-consistency algorithm [Sellmann 02] is used, the complexity of which is roughly $O(n(m + n \log n))$ where m is the total size of the domains.

The following constraint can be thought of as constraining n nodes in a graph to form a Hamiltonian circuit. The nodes are numbered from 1 to n. The circuit starts in node 1, visits each node, and returns to the origin.
circuit(+Succ)
circuit(+Succ, +Pred)
where Succ is a list of length \( n \) of domain variables or integers. The \( i \):th element of Succ (Pred) is the successor (predecessor) of \( i \) in the graph. True if the values form a Hamiltonian circuit.

The following constraint can be thought of as constraining \( n \) tasks, each with a start time \( S_j \) and a duration \( D_j \), so that no tasks ever overlap. The tasks can be seen as competing for some exclusive resource.

\[
\text{serialized(+Starts,+Durations)}
\]
\[
\text{serialized(+Starts,+Durations,+Options)}
\]
where \( \text{Starts} = [S_1, \ldots, S_n] \) and \( \text{Durations} = [D_1, \ldots, D_n] \) are lists of domain variables with finite bounds or integers. Durations must be non-negative. True if \( \text{Starts} \) and \( \text{Durations} \) denote a set of non-overlapping tasks, i.e.:

\[
\forall i, j \in [1, n] \land i < j :
\]
\[
S_i + D_i \leq S_j \lor \]
\[
S_j + D_j \leq S_i \lor \]
\[
D_i = 0 \lor \]
\[
D_j = 0
\]

The \text{serialized/[2,3]} constraint is merely a special case of \text{cumulative/[4,5]} (see below). Options is a list of zero or more of the following, where \text{Boolean} must be \text{true} or \text{false} (false is the default, except for the \text{bounds_only} option):

\text{precedences(Ps)}
\begin{itemize}
\item \text{d}(i, j, k), where \( i \) and \( j \) should be task numbers, and \( k \) should be a positive integer or \text{sup}, denoting:
\[
\begin{cases}
S_i + k \leq S_j \lor S_j \leq S_i, & \text{if } k \text{ is an integer} \\
S_j \leq S_i, & \text{if } k \text{ is sup}
\end{cases}
\]
\item \text{i-j in r}, where \( i \) and \( j \) should be task numbers, and \( r \) should be a \text{ConstantRange} (see Section 34.11.1 [Syntax of Indexicals], page 491), denoting:
\[
S_i - S_j = D_{ij} \land D_{ij} \in r
\]
\end{itemize}

\text{resource(R)}
\begin{itemize}
\item \( R \) is unified with a term that can be passed to \text{order_resource/2} (see Section 34.4 [Enumeration Predicates], page 465) in order to find a consistent ordering of the tasks.
path_consistency(Boolean)
    if true, a redundant path-consistency algorithm will be used inside
    the constraint in an attempt to improve the pruning.

static_sets(Boolean)
    if true, a redundant algorithm will be used, which reasons about
    the set of tasks that must precede (be preceded by) a given task,
    in an attempt to tighten the lower (upper) bound of a given start
    variable.

edge_finder(Boolean)
    if true, a redundant algorithm will be used, which attempts to
    identify tasks that necessarily precede or are preceded by some set
    of tasks.

decomposition(Boolean)
    if true, an attempt is made to decompose the constraint each time
    it is resumed.

bounds_only(Boolean)
    if true, the constraints will only prune the bounds of the \( S_i \)
    variables, and not inside the domains.

Whether it’s worthwhile to switch on any of the latter five options is highly
problem dependent.

serialized/3 can model a set of tasks to be serialized with sequence-dependent
setup times. For example, the following constraint models three tasks, all with
duration 5, where task 1 must precede task 2 and task 3 must either complete
before task 2 or start at least 10 time units after task 2 started:

```prolog
?- domain([S1,S2,S3], 0, 20),
    serialized([S1,S2,S3], [5,5,5],
                [precedences([d(2,1,sup),d(2,3,10)])]).
```

\( S1 \) in 0..15,
\( S2 \) in 5..20,
\( S3 \) in 0..20

The bounds of \( S1 \) and \( S2 \) changed because of the precedence constraint. Setting
\( S2 \) to 5 will propagate \( S1=0 \) and \( S3 \) in 15..20.

The following constraint can be thought of as constraining \( n \) tasks to be placed in time
and on \( m \) machines. Each machine has a resource limit, which is interpreted as a lower or
upper bound on the total amount of resource used on that machine at any point in time
that intersects with some task.

A task is represented by a term task(\( O_i, D_i, E_i, H_i, M_i \)) where \( O_i \) is the start time, \( D_i \) the
duration, \( E_i \) the end time, \( H_i \) the resource consumption, and \( M_i \) a machine identifier.

A machine is represented by a term machine(\( M_j, L_j \)) where \( M_j \) is the identifier and \( L_j \) is
the resource limit of the machine.
All fields are domain variables with bounded domains, or integers. \( L_j \) must be an integer. \( D_i \) must be non-negative, but \( H_i \) may be either positive or negative. Negative resource consumption is interpreted as resource production.

\[
cumulatives(+\text{Tasks},+\text{Machines})
cumulatives(+\text{Tasks},+\text{Machines},+\text{Options})
\]

Options is a list of zero or more of the following, where Boolean must be true or false (false is the default):

- **bound(B)** If lower (the default), each resource limit is treated as a lower bound. If upper, each resource limit is treated as an upper bound.
- **prune(P)** If all (the default), the constraint will try to prune as many variables as possible. If next, only variables that occur in the first non-ground task term (wrt. the order given when the constraint was posted) can be pruned.

- **generalization(Boolean)** If true, extra reasoning based on assumptions on machine assignment will be done to infer more.
- **task_intervals(Boolean)** If true, extra global reasoning will be performed in an attempt to infer more.

The following constraint can be thought of as constraining \( n \) tasks, each with a start time \( S_j \), a duration \( D_j \), and a resource amount \( R_j \), so that the total resource consumption does not exceed Limit at any time:

\[
cumulative(+\text{Starts},+\text{Durations},+\text{Resources},?\text{Limit})
cumulative(+\text{Starts},+\text{Durations},+\text{Resources},?\text{Limit},+\text{Options})
\]

where \( \text{Starts} = [S_1, \ldots, S_n] \), \( \text{Durations} = [D_1, \ldots, D_n] \), and \( \text{Resources} = [R_1, \ldots, R_n] \) are lists of domain variables with finite bounds or integers, and \( \text{Limit} \) is a domain variable with finite bounds or an integer. Durations, Resources and Limit must be non-negative. Let:

\[
\begin{align*}
a &= \min(S_1, \ldots, S_n) \\
b &= \max(S_1 + D_1, \ldots, S_n + D_n) \\
R_{ij} &= \begin{cases} R_j, & \text{if } S_j \leq i < S_j + D_j \\ 0, & \text{otherwise} \end{cases}
\end{align*}
\]

The constraint holds if:

\[
\forall a \leq i < b : R_{i1} + \cdots + R_{in} \leq \text{Limit}
\]

If given, Options should be of the same form as in `serialized/3`, except the resource(R) option is not useful in `cumulative/5`.

The `cumulative/4` constraint is due to Aggoun and Beldiceanu [Aggoun & Beldiceanu 93].
The following constraint captures the relation between a list of values, a list of the values in ascending order, and their positions in the original list:

$$\text{sorting}(+Xs,+Ps,+Ys)$$

where $Xs = [X_1, \ldots, X_n]$, $Ps = [P_1, \ldots, P_n]$, and $Ys = [Y_1, \ldots, Y_n]$ are lists of domain variables or integers. The constraint holds if the following are true:

- $Ys$ is in ascending order.
- $Ps$ is a permutation of $[1, n]$.
- $\forall i \in [1, n] : X_i = Y_{P_i}$

In practice, the underlying algorithm [Mehlhorn 00] is likely to achieve interval-consistency, and is guaranteed to do so if $Ps$ is ground or completely free.

The following constraints model a set or lines or rectangles, respectively, so that no pair of objects overlap:

$$\text{disjoint1}(+Lines)$$
$$\text{disjoint1}(+Lines,+Options)$$

where $Lines$ is a list of terms $F(S_j, D_j)$ or $F(S_j, D_j, T_j)$, $S_j$ and $D_j$ are domain variables with finite bounds or integers denoting the origin and length of line $j$ respectively, $F$ is any functor, and the optional $T_j$ is an atomic term denoting the type of the line. $T_j$ defaults to 0 (zero).

$Options$ is a list of zero or more of the following, where $Boolean$ must be $true$ or $false$ ($false$ is the default):

- $\text{decomposition}(Boolean)$
  
  if $true$, an attempt is made to decompose the constraint each time it is resumed.

- $\text{global}(Boolean)$
  
  if $true$, a redundant algorithm using global reasoning is used to achieve more complete pruning.

- $\text{wrap}(Min,Max)$
  
  If used, the space in which the lines are placed should be thought of as a circle where positions $Min$ and $Max$ coincide, where $Min$ and $Max$ should be integers. That is, the space wraps around. Furthermore, this option forces the domains of the origin variables to be inside $[Min,Max-1]$.

- $\text{margin}(T_1,T_2,D)$
  
  This option imposes a minimal distance $D$ between the end point of any line of type $T_1$ and the origin of any line of type $T_2$. $D$ should be a positive integer or $\text{sup}$. If $\text{sup}$ is used, all lines of type $T_2$ must be placed before any line of type $T_1$. This option interacts with the $\text{wrap}/2$ option in the sense that distances are counted with possible wrap-around, and the distance between any end point and origin is always finite.
The file `library('clpfd/examples/bridge.pl')` contains an example where `disjoint1/2` is used for scheduling non-overlapping tasks.

\[
\text{disjoint2(+Rectangles)}
\]
\[
\text{disjoint2(+Rectangles,+Options)}
\]

where `Rectangles` is a list of terms \(F(S_{j1}, D_{j1}, S_{j2}, D_{j2})\) or \(F(S_{j1}, D_{j1}, S_{j2}, D_{j2}, T_j)\), \(S_{j1}\) and \(D_{j1}\) are domain variables with finite bounds or integers denoting the origin and size of rectangle \(j\) in the X dimension, \(S_{j2}\) and \(D_{j2}\) are the values for the Y dimension, \(F\) is any functor, and the optional \(T_j\) is an atomic term denoting the type of the rectangle. \(T_j\) defaults to 0 (zero).

`Options` is a list of zero or more of the following, where `Boolean` must be `true` or `false` (false is the default):

\[
\text{decomposition(Boolean)}
\]

If `true`, an attempt is made to decompose the constraint each time it is resumed.

\[
\text{global(Boolean)}
\]

If `true`, a redundant algorithm using global reasoning is used to achieve more complete pruning.

\[
\text{wrap(Min1,Max1,Min2,Max2)}
\]

`Min1` and `Max1` should be either integers or the atoms `inf` and `sup` respectively. If they are integers, the space in which the rectangles are placed should be thought of as a cylinder wrapping around the X dimension where positions `Min1` and `Max1` coincide. Furthermore, this option forces the domains of the \(S_{j1}\) variables to be inside \([Min1,Max1-1]\).

`Min2` and `Max2` should be either integers or the atoms `inf` and `sup` respectively. If they are integers, the space in which the rectangles are placed should be thought of as a cylinder wrapping around the Y dimension where positions `Min2` and `Max2` coincide. Furthermore, this option forces the domains of the \(S_{j2}\) variables to be inside \([Min2,Max2-1]\).

If all four are integers, the space is a toroid wrapping around both dimensions.

\[
\text{margin(T1,T2,D1,D2)}
\]

This option imposes minimal distances \(D_1\) in the X dimension and \(D_2\) in the Y dimension between the end point of any rectangle of type \(T_1\) and the origin of any rectangle of type \(T_2\). \(D_1\) and \(D_2\) should be positive integers or `sup`. If `sup` is used, all rectangles of type \(T_2\) must be placed before any rectangle of type \(T_1\) in the relevant dimension.

This option interacts with the `wrap/4` option in the sense that distances are counted with possible wrap-around, and the distance between any end point and origin is always finite.

The file `library('clpfd/examples/squares.pl')` contains an example where `disjoint2/2` is used for tiling squares.
synchronization(Boolean)

Let the assignment dimension and the temporal dimension denote the two dimensions, no matter which is the X and which is the Y dimension. If Boolean is true, a redundant algorithm is used to achieve more complete pruning for the following case:

- All rectangles have size 1 in the assignment dimension.
- Some rectangles have the same origin and size in the temporal dimension, and that origin is not yet fixed.

The following example shows an artificial placement problem involving 25 rectangles including four groups of rectangles whose left and right borders must be aligned. If Synch is true, it can be solved with first-fail labeling in 23 backtracks. If Synch is false, 60 million backtracks do not suffice to solve it.

```prolog
ex([O1,Y1a,Y1b,Y1c,  
    O2,Y2a,Y2b,Y2c,Y2d,  
    O3,Y3a,Y3b,Y3c,Y3d,  
    O4,Y4a,Y4b,Y4c],  
   Synch) :-  
   domain([Y1a,Y1b,Y1c,  
            Y2a,Y2b,Y2c,Y2d,  
            Y3a,Y3b,Y3c,Y3d,  
            Y4a,Y4b,Y4c], 1, 5),  
   O1 in 1..28,  
   O2 in 1..26,  
   O3 in 1..22,  
   O4 in 1..25,  
   disjoint2([t(1,1,5,1), t(20,4,5,1),  
              t(1,1,4,1), t(14,4,4,1),  
              t(1,2,3,1), t(24,2,3,1),  
              t(1,2,2,1), t(21,1,2,1),  
              t(1,3,1,1), t(14,2,1,1),  
              t(01,3,Y1a,1), t(01,3,Y1b,1),  
              t(01,3,Y1c,1), t(02,5,Y2a,1),  
              t(02,5,Y2b,1), t(02,5,Y2c,1),  
              t(02,5,Y2d,1), t(03,9,Y3a,1),  
              t(03,9,Y3b,1), t(03,9,Y3c,1),  
              t(03,9,Y3d,1), t(04,6,Y4a,1),  
              t(04,6,Y4b,1), t(04,6,Y4c,1)],  
   [synchronization(Synch)]).
```
The following constraints express the fact that several vectors of domain variables are in ascending lexicographic order:

```
lex_chain(+Vectors)
lex_chain(+Vectors,+Options)
```

where `Vectors` is a list of vectors (lists) of domain variables with finite bounds or integers. The constraint holds if `Vectors` are in ascending lexicographic order. `Options` is a list of zero or more of the following:

```
op(Op)
```

If `Op` is the atom `#=<` (the default), the constraints holds if `Vectors` are in non-descending lexicographic order. If `Op` is the atom `#<`, the constraints holds if `Vectors` are in strictly ascending lexicographic order.

```
among(Least,Most,Values)
```

If given, `Least` and `Most` should be integers such that $0 \leq Least \leq Most$ and `Values` should be a list of distinct integers. This option imposes the additional constraint on each vector in `Vectors` that at least `Least` and at most `Most` elements should belong to `Values`.

In the absence of an `among/3` option, the underlying algorithm [Carlsson & Beldiceanu 02] guarantees domain-consistency.

### 34.3.5 User-Defined Constraints

New, primitive constraints can be added defined by the user on two different levels. On a higher level, constraints can be defined using the global constraint programming interface; see Section 34.8 [Defining Global Constraints], page 471. Such constraints can embody specialized algorithms and use the full power of Prolog. They cannot be reified.

On a lower level, new primitive constraints can be defined with indexicals. In this case, they take part in the basic constraint solving algorithm and express custom designed rules for special cases of the overall local propagation scheme. Such constraints are called FD predicates; see Section 34.9 [Defining Primitive Constraints], page 479. They can optionally be reified.

### 34.4 Enumeration Predicates

As is usually the case with finite domain constraint solvers, this solver is not complete. That is, it does not ensure that the set of posted constraints is satisfiable. One must resort to search (enumeration) to check satisfiability and get particular solutions.

The following predicates provide several variants of search:


\texttt{indomain(?X)}

where \(X\) is a domain variable with a bounded domain or an integer. Assigns, in increasing order via backtracking, a feasible value to \(X\).

\texttt{labeling(:Options, +Variables)}

where \(Variables\) is a list of domain variables or integers and \(Options\) is a list of search options. The domain variables must all have bounded domains. True if an assignment of the variables can be found, which satisfies the posted constraints.

\texttt{first_bound(+BB0, -BB)}

\texttt{later_bound(+BB0, -BB)}

Provides an auxiliary service for the \texttt{value(Enum)} option (see below).

\texttt{minimize(:Goal, ?X)}

\texttt{maximize(:Goal, ?X)}

Uses a branch-and-bound algorithm with restart to find an assignment that minimizes (maximizes) the domain variable \(X\). \texttt{Goal} should be a Prolog goal that constrains \(X\) to become assigned, and could be a \texttt{labeling/2} goal. The algorithm calls \texttt{Goal} repeatedly with a progressively tighter upper (lower) bound on \(X\) until a proof of optimality is obtained, at which time \texttt{Goal} and \(X\) are unified with values corresponding to the optimal solution.

The \texttt{Options} argument of \texttt{labeling/2} controls the order in which variables are selected for assignment (variable choice heuristic), the way in which choices are made for the selected variable (value choice heuristic), and whether all solutions or a single, optimal solution should be found. The options are divided into four groups. One option may be selected per group. Also, the number of assumptions (choices) made during the search can be collected. Finally, a discrepancy limit can be imposed.

The following options control the order in which the next variable is selected for assignment.

\texttt{leftmost} The leftmost variable is selected. This is the default.

\texttt{min} The leftmost variable with the smallest lower bound is selected.

\texttt{max} The leftmost variable with the greatest upper bound is selected.

\texttt{ff} The first-fail principle is used: the leftmost variable with the smallest domain is selected.

\texttt{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.

\texttt{variable(Sel)}

\(Sel\) is a predicate to select the next variable. Given \(Vars\), the variables that remain to label, it will be called as \(Sel(Vars, Selected, Rest)\).
Sel is expected to succeed determinately, unifying Selected and Rest with the selected variable and the remaining list, respectively.

Sel should be a callable term, optionally with a module prefix, and the arguments Vars,Selected,Rest will be appended to it. For example, if Sel is mod:sel(Param), it will be called as mod:sel(Param,Vars,Selected,Rest).

The following options control the way in which choices are made for the selected variable X:

- **step**: Makes a binary choice between \( X \neq B \) and \( X \neq B \), where \( B \) is the lower or upper bound of \( X \). This is the default.
- **enum**: Makes a multiple choice for \( X \) corresponding to the values in its domain.
- **bisect**: Makes a binary choice between \( X \leq M \) and \( X > M \), where \( M \) is the midpoint of the domain of \( X \). This strategy is also known as domain splitting.

\[ \text{value(Enum)} \]

Enum is a predicate that should narrow the domain of \( X \), possibly but not necessarily to a singleton. It will be called as \( \text{Enum}(X, \text{Rest}, BB0, BB) \) where \( \text{Rest} \) is the list of variables that need labeling except \( X \), and \( BB0 \) and \( BB \) are parameters described below.

Enum is expected to succeed nondeterminately, narrowing the domain of \( X \), and to backtrack one or more times, providing alternative narrowings. To ensure that branch-and-bound search works correctly, it must call the auxiliary predicate `first_bound(BB0, BB)` in its first solution. Similarly, it must call the auxiliary predicate `later_bound(BB0, BB)` in any alternative solution.

Enum should be a callable term, optionally with a module prefix, and the arguments \( X, \text{Rest}, BB0, BB \) will be appended to it. For example, if Enum is mod:enum(Param), it will be called as mod:enum(Param,X,Rest,BB0,BB).

The following options control the order in which the choices are made for the selected variable \( X \). Not useful with the `value(Enum)` option:

- **up**: The domain is explored in ascending order. This is the default.
- **down**: The domain is explored in descending order.

The following options control whether all solutions should be enumerated by backtracking or whether a single solution that minimizes (maximizes) \( X \) is returned, if one exists.

- **all**: All solutions are enumerated. This is the default.
minimize(X)
maximize(X)

Uses a branch-and-bound algorithm to find an assignment that minimizes (maximizes) the domain variable X. The labeling should constrain X to become assigned for all assignments of Variables. It is useful to combine this option with the time_out/2 option (see below).

The following option counts the number of assumptions (choices) made during the search:

assumptions(K)

When a solution is found, K is unified with the number of choices made.

A limit on the discrepancy of the search can be imposed:

discrepancy(D)

On the path leading to the solution there are at most D choicepoints in which a non-leftmost branch was taken.

Finally, a time limit on the search can be imposed:

time_out(Time,Flag)

This is equivalent to a goal time_out(labeling(...),Time,Flag) (see Chapter 49 [Timeout], page 767). Furthermore, if combined with the minimize(V) or maximize(V) option, and the time limit is reached, the values of Variables and V will be those of the best solution found.

For example, to enumerate solutions using a static variable ordering, use:

\[ \text{\texttt{| ?- constraints(Variables),}} \]
\[ \text{\texttt{labeling([], Variables).}} \]
\[ \text{\texttt{\%same as \[leftmost,step,up,all\]}} \]

To minimize a cost function using branch-and-bound search, a dynamic variable ordering using the first-fail principle, and domain splitting exploring the upper part of domains first, use:

\[ \text{\texttt{| ?- constraints(Variables, Cost),}} \]
\[ \text{\texttt{labeling([ff,bisect,down,minimize(Cost)], Variables).}} \]

The file library('clpfd/examples/tsp.pl') contains an example of user-defined variable and value choice heuristics.

As opposed to the predicates above, which search for consistent assignments to domain variables, the following predicate searches for a consistent ordering among tasks competing for an exclusive resource, without necessarily fixing their start times:
order_resource(+Options, +Resource)

where Options is a list of search options and Resource represents a resource as returned by serialized/3 (see Section 34.3.4 [Combinatorial Constraints], page 452) on which tasks must be serialized. True if a total ordering can be imposed on the tasks, enumerating all such orderings via backtracking.

The search options control the construction of the total ordering. It may contain at most one of the following atoms, selecting a strategy:

first The ordering is built by repetitively selecting some task to be placed before all others.

last The ordering is built by repetitively selecting some task to be placed after all others.

and at most one of the following atoms, controlling which task to select at each step. If first is chosen (the default), the task with the smallest value is selected; otherwise, the task with the greatest value is selected.

est The tasks are ordered by earliest start time.

lst The tasks are ordered by latest start time.

etc The tasks are ordered by earliest completion time.

lct The tasks are ordered by latest completion time.

[first,est] (the default) and [last,lct] can be good heuristics.

34.5 Statistics Predicates

The following predicates can be used to get execution statistics.

fd_statistics(?Key, ?Value)

This allows a program to access execution statistics specific to this solver. General statistics about CPU time and memory consumption etc. is available from the built-in predicate statistics/2.

For each of the possible keys Key, Value is unified with the current value of a counter, which is simultaneously zeroed. The following counters are maintained. See Section 34.7 [The Constraint System], page 470, for details of what they all mean:

resumptions The number of times a constraint was resumed.

entailments The number of times a (dis)entailment was detected by a constraint.

prunings The number of times a domain was pruned.

backtracks The number of times a contradiction was found by a domain being wiped out, or by a global constraint signalling failure. Other causes
of backtracking, such as failed Prolog tests, are not covered by this counter.

**constraints**
The number of constraints created.

**fd_statistics**
Displays on the standard error stream a summary of the above statistics. All counters are zeroed.

### 34.6 Answer Constraints

By default, the answer constraint only shows the projection of the store onto the variables that occur in the query, but not any constraints that may be attached to these variables, nor any domains or constraints attached to other variables. This is a conscious decision, as no efficient algorithm for projecting answer constraints onto the query variables is known for this constraint system.

It is possible, however, to get a complete answer constraint including all variables that took part in the computation and their domains and attached constraints. This is done by asserting a clause for the following predicate:

```
clpfd:full_answer
```

If false (the default), the answer constraint, as well as constraints projected by `clpfd:project_attributes/2`, `clpfd:attribute_goal/2` and their callers, only contain the domains of the query variables. If true, those constraints contain the domains and any attached constraints of all variables. Initially defined as a dynamic predicate with no clauses.

### 34.7 The Constraint System

#### 34.7.1 Definitions

The constraint system is based on domain constraints and indexicals. A domain constraint is an expression \( X :: I \), where \( X \) is a domain variable and \( I \) is a nonempty set of integers.

A set \( S \) of domain constraints is called a store. \( D(X, S) \), the domain of \( X \) in \( S \), is defined as the intersection of all \( I \) such that \( X :: I \) belongs to \( S \). The store is contradictory if the domain of some variable is empty; otherwise, it is consistent. A consistent store \( S' \) is an extension of a store \( S \) iff, for all variables \( X \), \( D(X, S') \) is contained in \( D(X, S) \).

The following definitions, adapted from [Van Hentenryck et al. 95], define important notions of consistency and entailment of constraints wrt. stores.

A ground constraint is true if it holds and false otherwise.
A constraint $C$ is domain-consistent wrt. $S$ iff, for each variable $X_i$ and value $V_i$ in $D(X_i, S)$, there exist values $V_j$ in $D(X_j, S)$, $1 \leq j \leq n \land i \neq j$, such that $C(V_1, \ldots, V_n)$ is true.

A constraint $C$ is domain-entailed by $S$ iff, for all values $V_j$ in $D(X_j, S)$, $1 \leq j \leq n$, $C(V_1, \ldots, V_n)$ is true.

Let $D'(X, S)$ denote the interval $[\min(D(X, S)), \max(D(X, S))]$.

A constraint $C$ is interval-consistent wrt. $S$ iff, for each variable $X_i$, there exist values $V_j$ and $W_j$ in $D'(X_j, S)$, $1 \leq j \leq n, i \neq j$, such that $C(V_1, \ldots, \min(D(X_i, S)), \ldots, V_n)$ and $C(W_1, \ldots, \max(D(X_i, S)), \ldots, W_n)$ are both true.

A constraint $C$ is interval-entailed by $S$ iff, for all values $V_j$ in $D'(X_j, S)$, $1 \leq j \leq n$, $C(V_1, \ldots, V_n)$ is true.

Finally, a constraint is domain-disentailed (interval-disentailed) by $S$ iff its negation is domain-entailed (interval-entailed) by $S$.

### 34.7.2 Pitfalls of Interval Reasoning

In most circumstances, arithmetic constraints maintain interval-consistency and detect interval-entailment and -disentailment. There are cases where an interval-consistency maintaining constraint may detect a contradiction when the constraint is not yet interval-disentailed, as the following example illustrates. Note that $X \neq Y$ maintains domain-consistency if both arguments are constants or variables:

```prolog
| ?- X+Y #\= Z, X=1, Z=6, Y in 1..10, Y \#\= 5. no |
```

Since $1+5\neq6$ holds, $X+Y \neq Z$ is not interval-disentailed, although any attempt to make it interval-consistent wrt. the store results in a contradictory store.

### 34.8 Defining Global Constraints

#### 34.8.1 The Global Constraint Programming Interface

This section describes a programming interface by means of which new constraints can be written. The interface consists of a set of predicates provided by this library module. Constraints defined in this way can take arbitrary arguments and may use any constraint
solving algorithm, provided it makes sense. Reification cannot be expressed in this interface; instead, reification may be achieved by explicitly passing a 0/1-variable to the constraint in question.

Global constraints have state, which may be updated each time the constraint is resumed. The state information may be used e.g. in incremental constraint solving.

The following two predicates are the principal entrypoints for defining and posting new global constraints:

\[
\text{clpfd:dispatch\_global}(+\text{Constraint}, +\text{State0}, -\text{State}, -\text{Actions}) \quad \text{extendible}
\]

Tells the solver how to solve constraints of the form \text{Constraint}. Defined as a dynamic, multifile predicate.

When defining a new constraint, a clause of this predicate must be added. Its body defines a constraint solving method and should always succeed determinately. When a global constraint is called or resumed, the solver will call this predicate to deal with the constraint.

**Please note:** the constraint is identified by its principal functor; there is no provision for having two constraints with the same name in different modules. It is good practice to include a cut in every clause of \text{clpfd:dispatch\_global}/4.

\text{State0} and \text{State} are the old and new state respectively.

The constraint solving method must not invoke the constraint solver recursively e.g. by binding variables or posting new constraints; instead, \text{Actions} should be unified with a list of requests to the solver. Each request should be of the following form:

- **exit** The constraint has become entailed, and ceases to exist.
- **fail** The constraint has become disentailed, causing the solver to backtrack.
- **X = V** The solver binds \text{X} to \text{V}.
- **X in R** The solver constrains \text{X} to be a member of the \text{ConstantRange} \text{R} (see Section 34.11.1 [Syntax of Indexicals], page 491).
- **X in\_set S** The solver constrains \text{X} to be a member of the FD set \text{S} (see Section 34.8.3 [FD Set Operations], page 475).

- **call(Goal)**
  The solver calls the goal or constraint \text{Goal}, which should be module prefixed unless it is a built-in predicate or an exported predicate of the \text{clpfd} module.

\text{Goal} is executed as any Prolog goal, but in a context where some constraints may already be enqueued for execution, in which case those constraints will run after the completion of the call request.
fd_global(:Constraint, +State, +Susp)
fd_global(:Constraint, +State, +Susp, +Options)

where Constraint is a constraint goal, State is its initial state, and Susp is a term encoding how the constraint should wake up in response to domain changes. This predicate posts the constraint.

Susp is a list of $F(Var)$ terms where Var is a variable to suspend on and $F$ is a functor encoding when to wake up:

- \( \text{dom}(X) \) wake up when the domain of $X$ has changed
- \( \text{min}(X) \) wake up when the lower bound of $X$ has changed
- \( \text{max}(X) \) wake up when the upper bound of $X$ has changed
- \( \text{minmax}(X) \) wake up when the lower or upper of $X$ has changed
- \( \text{val}(X) \) wake up when $X$ has become ground

Options is a list of zero or more of the following:

- source(Term)
  By default, the symbolic form computed by \( \text{fd_copy_term/3} \), and shown in the answer constraint if \( \text{clpfd:full_answer} \) holds, equals Constraint, module name expanded. With this option, the symbolic form will instead be Term. In particular, if Term equals true, the constraint will not appear in the Body argument of \( \text{fd_copy_term/3} \). This can be useful if you are posting some redundant (implied) constraint.

- idempotent(Boolean)
  If true (the default), the constraint solving method is assumed to be idempotent. That is, in the scope of \( \text{clpfd:dispatch_global/4} \), the solver will not check for the resumption conditions for the given constraint, while performing its Actions. If false, an action may well cause the solver to resume the constraint that produced the action.
  If a variable occurs more than once in a global constraint that is being posted, or due to a variable-variable unification, the solver will no longer trust the constraint solving method to be idempotent.

For an example of usage, see Section 34.8.4 [A Global Constraint Example], page 477.

The following predicate controls operational aspects of the constraint solver:

fd_flag(+FlagName, ?OldValue, ?NewValue)

OldValue is the value of the FD flag FlagName, and the new value of FlagName is set to NewValue. The possible FD flag names and values are:

- overflow Determines the behavior on integer overflow conditions. Possible values:
  - error Raises a representation error (the default).
fail  Silently fails.

debug  Controls the visibility of constraint propagation. Possible values are on and off (the default). For internal use by library(fdbg).

34.8.2 Reflection Predicates

The constraint solving method needs access to information about the current domains of variables. This is provided by the following predicates, which are all constant time operations.

`fd_var(?X)` Checks that X is currently an unbound variable that is known to the CLPFD solver.

`fd_min(?X, ?Min)` where X is a domain variable (or an integer). Min is unified with the smallest value in the current domain of X, i.e. an integer or the atom inf denoting minus infinity.

`fd_max(?X, ?Max)` where X is a domain variable (or an integer). Max is unified with the upper bound of the current domain of X, i.e. an integer or the atom sup denoting infinity.

`fd_size(?X, ?Size)` where X is a domain variable (or an integer). Size is unified with the size of the current domain of X, if the domain is bounded, or the atom sup otherwise.

`fd_set(?X, ?Set)` where X is a domain variable (or an integer). Set is unified with an FD set term denoting the internal representation of the current domain of X; see below.

`fd_dom(?X, ?Range)` where X is a domain variable (or an integer). Range is unified with a ConstantRange (see Section 34.11.1 [Syntax of Indexicals], page 491) denoting the current domain of X.

`fd_degree(?X, ?Degree)` where X is a domain variable (or an integer). Degree is unified with the number of constraints that are attached to X.

Please note: this number may include some constraints that have been detected as entailed. Also, Degree is not the number of neighbors of X in the constraint network.

The following predicates can be used for computing the set of variables that are (transitively) connected via constraints to some given variable(s).
fd_neighbors(+Var, -Neighbors)
   Given a domain variable Var, Neighbors is the set of variables that can be reached from Var via constraints posted so far.

fd_closure(+Vars, -Closure)
   Given a list Vars of domain variables, Closure is the set of variables (including Vars) that can be transitively reached via constraints posted so far. Thus, fd_closure/2 is the transitive closure of fd_neighbors/2.

The following predicate can be used for computing a symbolic form of the constraints that are transitively attached to some term. This is useful e.g. in the context of asserting or copying terms, as these operations are not supported on terms containing domain variables:

fd_copy_term(+Term, -Template, -Body)
   Given a term Term containing domain variables, Template is a copy of the same term with all variables renamed to new variables such that executing Body will post constraints equivalent to those that Term is attached to.

   For example:
   | ?- X in 0..1, Y in 10..11, X+5 #=< Y, 
   | fd_copy_term(f(X,Y), Template, Body).
   | Body = _A in_set[[0|1]], _B in_set[[10|11]], clpfd:'t>=u+c'(_B,_A,5),]
   | Template = f(_A,_B),
   | X in 0..1,
   | Y in 10..11

34.8.3 FD Set Operations

The domains of variables are internally represented compactly as FD set terms. The details of this representation are subject to change and should not be relied on. Therefore, a number of operations on FD sets are provided, as such terms play an important role in the interface. The following operations are the primitive ones:

is_fdset(+Set)
   Set is a valid FD set.

empty_fdset(?Set)
   Set is the empty FD set.

fdset_parts(?Set, ?Min, ?Max, ?Rest)
   Set is an FD set, which is a union of the non-empty interval [Min,Max] and the FD set Rest, and all elements of Rest are greater than Max+1. Min and Max are both integers or the atoms inf and sup, denoting minus and plus infinity, respectively. Either Set or all the other arguments must be ground.

The following operations can all be defined in terms of the primitive ones, but in most cases, a more efficient implementation is used:
empty_interval(+Min, +Max)
    [Min, Max] is an empty interval.

fdset_interval(?Set, ?Min, ?Max)
    Set is an FD set, which is the non-empty interval [Min, Max].

fdset_singleton(?Set, ?Elt)
    Set is an FD set containing Elt only. At least one of the arguments must be ground.

fdset_min(+Set, -Min)
    Min is the lower bound of Set.

fdset_max(+Set, -Min)
    Max is the upper bound of Set. This operation is linear in the number of intervals of Set.

fdset_size(+Set, -Size)
    Size is the cardinality of Set, represented as sup if Set is infinite. This operation is linear in the number of intervals of Set.

list_to_fdset(+List, -Set)
    Set is the FD set containing the elements of List. Slightly more efficient if List is ordered.

fdset_to_list(+Set, -List)
    List is an ordered list of the elements of Set, which must be finite.

range_to_fdset(+Range, -Set)
    Set is the FD set containing the elements of the ConstantRange (see Section 34.11.1 [Syntax of Indexicals], page 491) Range.

fdset_to_range(+Set, -Range)
    Range is a constant interval, a singleton constant set, or a union of such, denoting the same set as Set.

fdset_add_element(+Set1, +Elt, -Set2)
    Set2 is Set1 with Elt inserted in it.

fdset_del_element(+Set1, +Elt, -Set2)
    Set2 is like Set1 but with Elt removed.

fdset_disjoint(+Set1, +Set2)
    The two FD sets have no elements in common.

fdset_intersect(+Set1, +Set2)
    The two FD sets have at least one element in common.

fdset_intersection(+Set1, +Set2, -Intersection)
    Intersection is the intersection between Set1 and Set2.

fdset_intersection(+Sets, -Intersection)
    Intersection is the intersection of all the sets in Sets.

fdset_member(?Elt, +Set)
    is true when Elt is a member of Set. If Elt is unbound, Set must be finite.
fdset_eq(+Set1, +Set2)
  Is true when the two arguments represent the same set i.e. they are identical.

fdset_subset(+Set1, +Set2)
  Every element of Set1 appears in Set2.

fdset_subtract(+Set1, +Set2, -Difference)
  Difference contains all and only the elements of Set1 that are not also in Set2.

fdset_union(+Set1, +Set2, -Union)
  Union is the union of Set1 and Set2.

fdset_union(+Sets, -Union)
  Union is the union of all the sets in Sets.

fdset_complement(+Set, -Complement)
  Complement is the complement of Set wrt. inf..sup.

### 34.8.4 A Global Constraint Example

The following example defines a new global constraint `exactly(X, L, N)`, which is true if X occurs exactly N times in the list L of integers and domain variables. N must be an integer when the constraint is posted. A version without this restriction and defined in terms of reified equalities was presented earlier; see Section 34.2.3 [Reified Constraints], page 449.

This example illustrates the use of state information. The state has two components: the list of variables that could still be X, and the number of variables still required to be X.

The constraint is defined to wake up on any domain change.
% exactly.pl

/*
An implementation of exactly(I, X[1]...X[m], N):

Necessary condition: 0 =< N =< m.
Rewrite rules:

[1] |= X[i]=I  ⊢→ exactly(I, X[1]...X[i-1],X[i+1]...X[m], N-1):
[2] |= X[i]=I  ⊢→ exactly(I, X[1]...X[i-1],X[i+1]...X[m], N):
[3] |= N=0     ⊢→ X[1]=I ... X[m]=I
[4] |= N=m     ⊢→ X[1]=I ... X[m]=I
*/
:- use_module(library(clpfd)).

% the entrypoint
exact(I, Xs, N) :-
    dom_suspensions(Xs, Susp),
    fd_global(exactly(I,Xs,N), state(Xs,N), Susp).

dom_suspensions([], []).
dom_suspensions([X|Xs], [dom(X)|Susp]) :-
    dom_suspensions(Xs, Susp).

% the solver method
:- multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(exactly(I,_,_), state(Xs0,NO), state(Xs,N), Actions) :-
    exactly_solver(I, Xs0, Xs, NO, N, Actions).

exactly_solver(I, Xs0, Xs, NO, N, Actions) :-
    ex_filter(Xs0, Xs, NO, N, I),
    length(Xs, M),
    ( N:=0  -> Actions = [exit|Ps], ex_neq(Xs, I, Ps)
    ; N:=M  -> Actions = [exit|Ps], ex_eq(Xs, I, Ps)
    ; N>0, N<M -> Actions = []
    ; Actions = [fail]
    ).
Chapter 34: Constraint Logic Programming over Finite Domains

% exactly.pl

% rules [1,2]: filter the X’s, decrementing N
ex_filter([], [], N, N, _).
ex_filter([X|Xs], Ys, L, N, I) :- X==I, !,
    M is L-1,
    ex_filter(Xs, Ys, M, N, I).
ex_filter([X|Xs], Ys0, L, N, I) :-
    fd_set(X, Set),
    fdset_member(I, Set), !,
    Ys0 = [X|Ys],
    ex_filter(Xs, Ys, L, N, I).
ex_filter([_|Xs], Ys, L, N, I) :-
    ex_filter(Xs, Ys, L, N, I).

% rule [3]: all must be neq I
ex_neq(Xs, I, Ps) :-
    fdset_singleton(Set0, I),
    fdset_complement(Set0, Set),
    eq_all(Xs, Set, Ps).

% rule [4]: all must be eq I
ex_eq(Xs, I, Ps) :-
    fdset_singleton(Set, I),
    eq_all(Xs, Set, Ps).

eq_all([], _, []).

end_of_file.

% sample queries: | ?- exactly(5,[A,B,C],1), A=5. A = 5, B
| in(inf..4)\(6..sup), C in(inf..4)\(6..sup)

| ?- exactly(5,[A,B,C],1), A in 1..2, B in 3..4.
| C = 5,
| A in 1..2,
| B in 3..4

34.9 Defining Primitive Constraints

Indexicals are the principal means of defining constraints, but it is usually not necessary
to resort to this level of programming—most commonly used constraints are available in
a library and/or via macro-expansion. The key feature about indexicals is that they give
the programmer precise control over aspects of the operational semantics of the constraints.
Trade-offs can be made between the computational cost of the constraints and their pruning power. The indexical language provides many degrees of freedom for the user to select the level of consistency to be maintained depending on application-specific needs.

### 34.9.1 Indexicals

An indexical is a reactive functional rule of the form \( X \text{ in } R \), where \( R \) is a set valued range expression (see below). See Section 34.11.1 [Syntax of Indexicals], page 491, for a grammar defining indexicals and range expressions.

Indexicals can play one of two roles: propagating indexicals are used for constraint solving, and checking indexicals are used for entailment checking. When a propagating indexical fires, \( R \) is evaluated in the current store \( S \), which is extended by adding the new domain constraint \( X :: S(R) \) to the store, where \( S(R) \) denotes the value of \( R \) in \( S \). When a checking indexical fires, it checks if \( D(X, S) \) is contained in \( S(R) \), and if so, the constraint corresponding to the indexical is detected as entailed.

### 34.9.2 Range Expressions

A range expression has one of the following forms, where \( R_i \) denote range expressions, \( T_i \) denote integer valued term expressions, \( S(T_i) \) denotes the integer value of \( T_i \) in \( S \), \( X \) denotes a variable, \( I \) denotes an integer, and \( S \) denotes the current store.

- \( \text{dom}(X) \) evaluates to \( D(X, S) \)
- \( \{T_1, \ldots, T_n\} \) evaluates to \( \{S(T_1), \ldots, S(T_n)\} \). Any \( T_i \) containing a variable that is not “quantified” by \( \text{unionof}/3 \) will cause the indexical to suspend until this variable has been assigned.
- \( T_1..T_2 \) evaluates to the interval between \( S(T_1) \) and \( S(T_2) \).
- \( R_1 \cap R_2 \) evaluates to the intersection of \( S(R_1) \) and \( S(R_2) \)
- \( R_1 \cup R_2 \) evaluates to the union of \( S(R_1) \) and \( S(R_2) \)
- \( \neg R_2 \) evaluates to the complement of \( S(R_2) \)
- \( R_1 + R_2 \) evaluates to \( S(R_2) \) or \( S(T_2) \) added pointwise to \( S(R_1) \)
- \( R_1 \mod R_2 \) evaluates to \( S(R_1) \) pointwise modulo \( S(R_2) \) or \( S(T_2) \)
Constraint Logic Programming over Finite Domains

\( R_1 ? R_2 \) evaluates to \( S(R_2) \) if \( S(R_1) \) is a non-empty set; otherwise, evaluates to the empty set. This expression is commonly used in the context \( (R_1 ? (\text{inf..sup}) \vee R_3) \), which evaluates to \( S(R_3) \) if \( S(R_1) \) is an empty set; otherwise, evaluates to \( \text{inf..sup} \). As an optimization, \( R_3 \) is not evaluated while the value of \( R_1 \) is a non-empty set.

\[ \text{unionof}(X, R_1, R_2) \]

evaluates to the union of \( S(E_1), \ldots, S(E_N) \), where each \( E_i \) has been formed by substituting \( K \) for \( X \) in \( R_2 \), where \( K \) is the \( i \)th element of \( S(R_1) \). See Section 34.10.2 [N Queens], page 488, for an example of usage.

**Please note:** if \( S(R_1) \) is infinite, the evaluation of the indexical will be abandoned, and the indexical will simply suspend.

\[ \text{switch}(T, \text{MapList}) \]

evaluates to \( S(E) \) if \( S(T_1) \) equals \( K \) and \( \text{MapList} \) contains a pair \( K-E \). Otherwise, evaluates to the empty set. If \( T \) contains a variable that is not “quantified” by \( \text{unionof}/3 \), the indexical will suspend until this variable has been assigned.

When used in the body of an FD predicate (see Section 34.9.8 [Goal Expanded Constraints], page 486), a \( \text{relation}/3 \) expression expands to two indexicals, each consisting of a \( \text{switch}/2 \) expression nested inside a \( \text{unionof}/3 \) expression. Thus, the following constraints are equivalent:

\[ p(X, Y) +: \text{relation}(X, [1-{1}, 2-{1,2}, 3-{1,2,3}], Y). \]

\[ q(X, Y) +: \]

\[ X \text{ in unionof}(B, \text{dom}(Y), \text{switch}(B, [1-{1,2,3}, 2-{2,3}, 3-{3}]))), \]

\[ Y \text{ in unionof}(B, \text{dom}(X), \text{switch}(B, [1-{1}, 2-{1,2}, 3-{1,2,3}])). \]

### 34.9.3 Term Expressions

A term expression has one of the following forms, where \( T_1 \) and \( T_2 \) denote term expressions, \( X \) denotes a variable, \( I \) denotes an integer, and \( S \) denotes the current store.

\[ \text{min}(X) \] evaluates to the minimum of \( D(X, S) \)

\[ \text{max}(X) \] evaluates to the maximum of \( D(X, S) \)

\[ \text{card}(X) \] evaluates to the size of \( D(X, S) \)

\( X \) evaluates to the integer value of \( X \). The indexical will suspend until \( X \) is assigned.

\( I \) an integer

\( \text{inf} \) minus infinity

\( \text{sup} \) plus infinity

\( -T_1 \) evaluates to \( S(T_1) \) negated
$T_1 + T_2$ evaluates to the sum of $S(T_1)$ and $S(T_2)$

$T_1 - T_2$ evaluates to the difference of $S(T_1)$ and $S(T_2)$

$T_1 * T_2$ evaluates to the product of $S(T_1)$ and $S(T_2)$, where $S(T_2)$ must not be negative

$T_1 > T_2$ evaluates to the quotient of $S(T_1)$ and $S(T_2)$, rounded up, where $S(T_2)$ must be positive

$T_1 < T_2$ evaluates to the quotient of $S(T_1)$ and $S(T_2)$, rounded down, where $S(T_2)$ must be positive

$T_1 \mod T_2$ evaluates to the modulo of $S(T_1)$ and $S(T_2)$

### 34.9.4 Monotonicity of Indexicals

A range $R$ is monotone in $S$ iff the value of $R$ in $S'$ is contained in the value of $R$ in $S$, for every extension $S'$ of $S$. A range $R$ is anti-monotone in $S$ iff the value of $R$ in $S$ is contained in the value of $R$ in $S'$, for every extension $S'$ of $S$. By abuse of notation, we will say that $X$ in $R$ is (anti-)monotone iff $R$ is (anti-)monotone.

The consistency or entailment of a constraint $C$ expressed as indexicals $X$ in $R$ in a store $S$ is checked by considering the relationship between $D(X, S)$ and $S(R)$, together with the (anti-)monotonicity of $R$ in $S$. The details are given in Section 34.9.6 [Execution of Propagating Indexicals], page 485 and Section 34.9.7 [Execution of Checking Indexicals], page 486.

The solver checks (anti-)monotonicity by requiring that certain variables occurring in the indexical be ground. This sufficient condition can sometimes be false for an (anti-)monotone indexical, but such situations are rare in practice.

### 34.9.5 FD Predicates

The following example defines the constraint $X + Y = T$ as an FD predicate in terms of three indexicals. Each indexical is a rule responsible for removing values detected as incompatible from one particular constraint argument. Indexicals are not Prolog goals; thus, the example does not express a conjunction. However, an indexical may make the store contradictory, in which case backtracking is triggered:

```prolog
plus(X,Y,T) +:
    X in min(T) - max(Y) .. max(T) - min(Y),
    Y in min(T) - max(X) .. max(T) - min(X),
    T in min(X) + min(Y) .. max(X) + max(Y).
```

The above definition contains a single clause used for constraint solving. The first indexical wakes up whenever the bounds of $S(T)$ or $S(Y)$ are updated, and removes from $D(X, S)$ any values that are not compatible with the new bounds of $T$ and $Y$. Note that in the event of "holes" in the domains of $T$ or $Y$, $D(X, S)$ may contain some values that are incompatible
with \(X + Y = T\) but go undetected. Like most built-in arithmetic constraints, the above
definition maintains interval-consistency, which is significantly cheaper to maintain than
domain-consistency and suffices in most cases. The constraint could for example be used
as follows:

\[
\begin{align*}
| ?- X & \in 1..5, Y \in 2..8, \text{plus}(X,Y,T). \\
X & \in 1..5, \\
Y & \in 2..8, \\
T & \in 3..13
\end{align*}
\]

Thus, when an FD predicate is called, the ‘+:’ clause is activated.

The definition of a user constraint has to specify what domain constraints should be added
to the constraint store when the constraint is posted. Therefore the FD predicate contains a
set of indexicals, each representing a domain constraint to be added to the constraint store.
The actual domain constraint depends on the constraint store itself. For example, the third
indexical in the above FD predicate prescribes the domain constraint ‘\(T :: 3..13\)’ if the
store contains ‘\(X :: 1..5, Y :: 2..8\)’. As the domain of some variables gets narrower, the
indexical may enforce a new, stricter constraint on some other variables. Therefore such an
indexical (called a propagating indexical) can be viewed as an agent reacting to the changes
in the store by enforcing further changes in the store.

In general there are three stages in the lifetime of a propagating indexical. When it is posted
it may not be evaluated immediately (e.g. has to wait until some variables are ground before
being able to modify the store). Until the preconditions for the evaluation are satisfied, the
agent does not enforce any constraints. When the indexical becomes evaluable the resulting
domain constraint is added to the store. The agent then waits and reacts to changes in
the domains of variables occurring in the indexical by re-evaluating it and adding the new,
stricter constraint to the store. Eventually the computation reaches a phase when no further
refinement of the store can result in a more precise constraint (the indexical is entailed by
the store), and then the agent can cease to exist.

A necessary condition for the FD predicate to be correctly defined is the following: for any
store mapping each variable to a singleton domain the execution of the indexicals should
succeed without contradiction exactly when the predicate is intended to be true.

There can be several alternative definitions for the same user constraint with different
strengths in propagation. For example, the definition of \texttt{plusd} below encodes the same
\(X+Y=T\) constraint as the \texttt{plus} predicate above, but maintaining domain-consistency:
plusd(X,Y,T) +:
   X in dom(T) - dom(Y),
   Y in dom(T) - dom(X),
   T in dom(X) + dom(Y).

| ?- X in \{1\} \{3\}, Y in \{10\} \{20\}, plusd(X, Y, T).
X in\{1\} \{3\},
Y in\{10\} \{20\},
T in\{11\} \{13\} \{21\} \{23\}

This costs more in terms of execution time, but gives more precise results. For singleton
domains plus and plusd behave in the same way.

In our design, general indexicals can only appear in the context of FD predicate definitions.
The rationale for this restriction is the need for general indexicals to be able to suspend
and resume, and this ability is only provided by the FD predicate mechanism.

If the program merely posts a constraint, it suffices for the definition to contain a single
clause for solving the constraint. If a constraint is reified or occurs in a propositional
formula, the definition must contain four clauses for solving and checking entailment of the
constraint and its negation. The role of each clause is reflected in the “neck” operator.
The following table summarizes the different forms of indexical clauses corresponding to
a constraint C. In all cases, Head should be a compound term with all arguments being
distinct variables:

**Head +: Indexicals.**
The clause consists of propagating indexicals for solving C.

**Head -: Indexicals.**
The clause consists of propagating indexicals for solving the negation of C.

**Head +? Indexical.**
The clause consists of a single checking indexical for testing entailment of C.

**Head -? Indexical.**
The clause consists of a single checking indexical for testing entailment of the
negation of C.

When a constraint is reified, the solver spawns two reactive agents corresponding to detecting
entailment and disentailment. Eventually, one of them will succeed in this and consequently
will bind B to 0 or 1. A third agent is spawned, waiting for B to become assigned,
at which time the constraint (or its negation) is posted. In the mean time, the constraint
may have been detected as (dis)entailed, in which case the third agent is dismissed. The
waiting is implemented by means of the coroutining facilities of SICStus Prolog.

As an example of a constraint with all methods defined, consider the following library
constraint defining a disequation between two domain variables:
Chapter 34: Constraint Logic Programming over Finite Domains

\[\begin{align*}
'x\neq y'(X,Y) +: & \\
& X \text{ in } \{Y\}, \\
& Y \text{ in } \{X\}. \\
'x\neq y'(X,Y) -: & \\
& X \text{ in } \text{dom}(Y), \\
& Y \text{ in } \text{dom}(X). \\
'x\neq y'(X,Y) +? & \\
& X \text{ in } \text{\textbackslash}\text{dom}(Y). \\
'x\neq y'(X,Y) -? & \\
& X \text{ in } \{Y\}. \\
\end{align*}\]

The following sections provide more precise coding rules and operational details for indexicals. \(X \text{ in } R\) denotes an indexical corresponding to a constraint \(C\). \(S\) denotes the current store.

### 34.9.6 Execution of Propagating Indexicals

Consider the definition of a constraint \(C\) containing a propagating indexical \(X \text{ in } R\). Let \(TV(X, C, S)\) denote the set of values for \(X\) that can make \(C\) true in some ground extension of the store \(S\). Then the indexical should obey the following coding rules:

- all arguments of \(C\) except \(X\) should occur in \(R\)
- if \(R\) is ground in \(S\), \(S(R) = TV(X, C, S)\)

If the coding rules are observed, \(S(R)\) can be proven to contain \(TV(X, C, S)\) for all stores in which \(R\) is monotone. Hence it is natural for the implementation to wait until \(R\) becomes monotone before admitting the propagating indexical for execution. The execution of \(X \text{ in } R\) thus involves the following:

- If \(D(X, S)\) is disjoint from \(S(R)\), a contradiction is detected.
- If \(D(X, S)\) is contained in \(S(R)\), \(D(X, S)\) does not contain any values known to be incompatible with \(C\), and the indexical suspends, unless \(R\) is ground in \(S\), in which case \(C\) is detected as entailed.
- Otherwise, \(D(X, S)\) contains some values that are known to be incompatible with \(C\). Hence, \(X : : S(R)\) is added to the store (\(X\) is pruned), and the indexical suspends, unless \(R\) is ground in \(S\), in which case \(C\) is detected as entailed.

A propagating indexical is scheduled for execution as follows:

- it is evaluated initially as soon as it has become monotone
- it is re-evaluated when one of the following conditions occurs:
  1. the domain of a variable \(Y\) that occurs as \(\text{dom}(Y)\) or \(\text{card}(Y)\) in \(R\) has been updated
  2. the lower bound of a variable \(Y\) that occurs as \(\text{min}(Y)\) in \(R\) has been updated
  3. the upper bound of a variable \(Y\) that occurs as \(\text{max}(Y)\) in \(R\) has been updated
34.9.7 Execution of Checking Indexicals

Consider the definition of a constraint $C$ containing a checking indexical $X$ in $R$. Let $FV(X, C, S)$ denote the set of values for $X$ that can make $C$ false in some ground extension of the store $S$. Then the indexical should obey the following coding rules:

- all arguments of $C$ except $X$ should occur in $R$
- if $R$ is ground in $S$, $S(R) = TV(X, C, S)$

If the coding rules are observed, $S(R)$ can be proven to exclude $FV(X, C, S)$ for all stores in which $R$ is anti-monotone. Hence it is natural for the implementation to wait until $R$ becomes anti-monotone before admitting the checking indexical for execution. The execution of $X$ in $R$ thus involves the following:

- If $D(X, S)$ is contained in $S(R)$, none of the possible values for $X$ can make $C$ false, and so $C$ is detected as entailed.
- Otherwise, if $D(X, S)$ is disjoint from $S(R)$ and $R$ is ground in $S$, all possible values for $X$ will make $C$ false, and so $C$ is detected as disentailed.
- Otherwise, $D(X, S)$ contains some values that could make $C$ true and some that could make $C$ false, and the indexical suspends.

A checking indexical is scheduled for execution as follows:

- it is evaluated initially as soon as it has become anti-monotone
- it is re-evaluated when one of the following conditions occurs:
  1. the domain of $X$ has been pruned, or $X$ has been assigned
  2. the domain of a variable $Y$ that occurs as $\text{dom}(Y)$ or $\text{card}(Y)$ in $R$ has been pruned
  3. the lower bound of a variable $Y$ that occurs as $\text{min}(Y)$ in $R$ has been increased
  4. the upper bound of a variable $Y$ that occurs as $\text{max}(Y)$ in $R$ has been decreased

34.9.8 Goal Expanded Constraints

The arithmetic, membership, and propositional constraints described earlier are transformed at compile time to conjunctions of goals of library constraints.

Sometimes it is necessary to postpone the expansion of a constraint until runtime, e.g. if the arguments are not instantiated enough. This can be achieved by wrapping $\text{call}/1$ around the constraint.

Although space economic (linear in the size of the source code), the expansion of a constraint to library goals can have an overhead compared to expressing the constraint in terms of indexicals. Temporary variables holding intermediate values may have to be introduced, and the grain size of the constraint solver invocations can be rather small. The translation
of constraints to library goals has been greatly improved in the current version, so these
problems have virtually disappeared. However, for backward compatibility, an implementa-
tion by compilation to indexicals of the same constraints is also provided. An FD predicate
may be defined by a single clause:

\[
\text{Head } +: \text{ Constraint.}
\]

where Constraint is an arithmetic constraint or an element/3 or a relation/3 constraint.
This translation is only available for ‘+:’ clauses; thus, Head cannot be reified.

In the case of arithmetic constraints, the constraint must be over linear terms (see Sec-
tion 34.11.1 [Syntax of Indexicals], page 491). The memory consumption of the FD pred-
icate will be quadratic in the size of the source code. The alternative version of sum/8 in
Section 34.10.1 [Send More Money], page 487 illustrates this technique.

In the case of element(X, L, Y) or relation(X, L, Y), the memory consumption of the FD
predicate will be linear in the size of the source code. The execution time of the initial
evaluation of the FD predicate will be linear in the size of the initial domains for X and Y;
if these domains are infinite, no propagation will take place.

34.10 Example Programs

This section contains a few example programs. The first two programs are included in a
benchmark suite that comes with the distribution. The benchmark suite is run by typing:

```
| ?- compile(library('clpfd/examples/bench')).
| ?- bench.
```

34.10.1 Send More Money

Let us return briefly to the Send More Money problem (see Section 34.2.2 [A Constraint
Satisfaction Problem], page 448). Its sum/8 predicate will expand to a space-efficient con-
junction of library constraints. A faster but more memory consuming version is defined
simply by changing the neck symbol of sum/8 from ‘:-’ to ‘+:', thus turning it into an FD predicate:

\[
\text{sum}(S, E, N, D, M, O, R, Y) +:
\]

\[
\begin{align*}
& 1000* S + 100* E + 10* N + D \\
+ & 1000* M + 100* O + 10* R + E \\
\# = & 10000* M + 1000* O + 100* N + 10* E + Y.
\end{align*}
\]
34.10.2 N Queens

The problem is to place N queens on an NxN chess board so that no queen is threatened by another queen.

The variables of this problem are the N queens. Each queen has a designated row. The problem is to select a column for it.

The main constraint of this problem is that no queen threaten another. This is encoded by the no_threat/3 constraint and holds between all pairs \((X, Y)\) of queens. It could be defined as

\[
\text{no\_threat}(X, Y, I) :-
\begin{align*}
X & \neq Y, \\
X+I & \neq Y, \\
X-I & \neq Y.
\end{align*}
\]

However, this formulation introduces new temporary domain variables and creates twelve fine-grained indexicals. Worse, the disequalities only maintain interval-consistency and so may miss some opportunities for pruning elements in the middle of domains.

A better idea is to formulate no_threat/3 as an FD predicate with two indexicals, as shown in the program below. This constraint will not fire until one of the queens has been assigned (the corresponding indexical does not become monotone until then). Hence, the constraint is still not as strong as it could be.

For example, if the domain of one queen is 2..3, then it will threaten any queen placed in column 2 or 3 on an adjacent row, no matter which of the two open positions is chosen for the first queen. The commented out formulation of the constraint captures this reasoning, and illustrates the use of the unionof/3 operator. This stronger version of the constraint indeed gives less backtracking, but is computationally more expensive and does not pay off in terms of execution time, except possibly for very large chess boards.

It is clear that no_threat/3 cannot detect any incompatible values for a queen with domain of size greater than three. This observation is exploited in the third version of the constraint.

The first-fail principle is appropriate in the enumeration part of this problem.
:- use_module(library(clpfd)).

queens(N, L, LabelingType) :-
    length(L, N),
    domain(L, 1, N),
    constrain_all(L),
    labeling(LabelingType, L).

constrain_all([]).
constrain_all([X|Xs]) :-
    constrain_between(X, Xs, 1),
    constrain_all(Xs).

constrain_between(_, [], _N).
constrain_between(X, [Y|Ys], N) :-
    no_threat(X, Y, N),
    N1 is N+1,
    constrain_between(X, Ys, N1).

% version 1: weak but efficient
no_threat(X, Y, I) +:
    X in \({Y} \lor {Y+I} \lor {Y-I})),
    Y in \({X} \lor {X+I} \lor {X-I})).

/*
% version 2: strong but very inefficient version
no_threat(X, Y, I) +:
    X in \(\text{unionof}(B, \text{dom}(Y)) \lor (\{B\} \lor \{B+I\} \lor \{B-I})\)),
    Y in \(\text{unionof}(B, \text{dom}(X)) \lor (\{B\} \lor \{B+I\} \lor \{B-I})\)).

% version 3: strong but somewhat inefficient version
no_threat(X, Y, I) +:
    X in \(\text{unionof}(B, \text{dom}(Y)) \lor (\{B\} \lor \{B+I\} \lor \{B-I})\)),
    Y in \(\text{unionof}(B, \text{dom}(X)) \lor (\{B\} \lor \{B+I\} \lor \{B-I})\)).
*/

| ?- queens(8, L, [ff]).
L = [1,5,8,6,3,7,2,4]
34.10.3 Cumulative Scheduling

This example is a very small scheduling problem. We consider seven tasks where each task has a fixed duration and a fixed amount of used resource:

<table>
<thead>
<tr>
<th>Task</th>
<th>Duration</th>
<th>Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>t2</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>t3</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>t4</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>t5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>t6</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>t7</td>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>

The goal is to find a schedule that minimizes the completion time for the schedule while not exceeding the capacity 13 of the resource. The resource constraint is succinctly captured by a cumulative/4 constraint. Branch-and-bound search is used to find the minimal completion time.

This example was adapted from [Beldiceanu & Contejean 94].

```prolog
:- use_module(library(clpfd)).
:- use_module(library(lists), [append/3]).

schedule(Ss, End) :-
    length(Ss, 7),
    Ds = [16, 6,13, 7, 5,18, 4],
    Rs = [2, 9, 3, 7,10, 1,11],
    domain(Ss, 1, 30),
    domain([End], 1, 50),
    after(Ss, Ds, End),
    cumulative(Ss, Ds, Rs, 13),
    append(Ss, [End], Vars),
    labeling([minimize(End)], Vars). % label End last

after([], [], _).
after([S|Ss], [D|Ds], E) :- E #>= S+D, after(Ss, Ds, E).
```

% End of file

| ?- schedule(Ss, End).
Ss = [1,17,10,10,5,5,1],
End = 23

34.11 Syntax Summary
34.11.1 Syntax of Indexicals

\[
\begin{align*}
X & \ ::= \ variable \quad \{ \text{domain variable} \} \\
\text{Constant} & \ ::= \ integer \\
& \quad | \ inf \quad \{ \text{minus infinity} \} \\
& \quad | \ sup \quad \{ \text{plus infinity} \} \\
\text{Term} & \ ::= \ Constant \\
& \quad | \ X \quad \{ \text{suspend until assigned} \} \\
& \quad | \ min(X) \quad \{ \text{min. of domain of } X \} \\
& \quad | \ max(X) \quad \{ \text{max. of domain of } X \} \\
& \quad | \ card(X) \quad \{ \text{size of domain of } X \} \\
& \quad | \ - \ Term \\
& \quad | \ Term + Term \\
& \quad | \ Term - Term \\
& \quad | \ Term * Term \\
& \quad | \ Term \div Term \quad \{ \text{division rounded up} \} \\
& \quad | \ Term < Term \quad \{ \text{division rounded down} \} \\
& \quad | \ Term \mod Term \\
\text{TermSet} & \ ::= \ \{ \text{Term,} \ldots , \text{Term} \} \\
\text{Range} & \ ::= \ \text{TermSet} \\
& \quad | \ dom(X) \quad \{ \text{domain of } X \} \\
& \quad | \ Term .. Term \quad \{ \text{interval} \} \\
& \quad | \ Range \div Range \quad \{ \text{intersection} \} \\
& \quad | \ Range \ \setminus \ Range \quad \{ \text{union} \} \\
& \quad | \ \setminus \ Range \quad \{ \text{complement} \} \\
& \quad | \ - \ Range \quad \{ \text{pointwise negation} \} \\
& \quad | \ Range + Range \quad \{ \text{pointwise addition} \} \\
& \quad | \ Range - Range \quad \{ \text{pointwise subtraction} \} \\
& \quad | \ Range \mod Range \quad \{ \text{pointwise modulo} \} \\
& \quad | \ Range + Term \quad \{ \text{pointwise addition} \} \\
& \quad | \ Range - Term \quad \{ \text{pointwise subtraction} \} \\
& \quad | \ Term - Range \quad \{ \text{pointwise subtraction} \} \\
& \quad | \ Range \mod Term \quad \{ \text{pointwise modulo} \} \\
& \quad | \ Range ? Range \\
& \quad | \ unionof(X,Range,Range) \\
& \quad | \ switch(Term,\text{MapList}) \\
\text{ConstantSet} & \ ::= \ \{ \text{integer,} \ldots , \text{integer} \} \\
\text{ConstantRange} & \ ::= \ \text{ConstantSet} \\
& \quad | \ Constant .. Constant \\
& \quad | \ ConstantRange \ \setminus \\
& \quad | \ ConstantRange \ \setminus \\
& \quad | \ \setminus \ ConstantRange \\
\text{MapList} & \ ::= \ []
\end{align*}
\]
492 SICStus Prolog

\[ \text{\emph{CList}} ::= [\, ] \]

\[ \text{\emph{Indexical}} ::= X \text{ in } \text{Range} \]
\[ \text{\emph{Indexicals}} ::= \text{\emph{Indexical}} \mid \text{\emph{Indexical}, \emph{Indexicals}} \]

\[ \text{\emph{ConstraintBody}} ::= \text{\emph{Indexicals}} \mid \text{LinExpr} \text{ RelOp } \text{LinExpr} \mid \text{element}(X, \text{CList}, X) \mid \text{relation}(X, \text{MapList}, X) \]

\[ \text{\emph{Head}} ::= \text{term} \{ \text{a compound term with unique variable args} \} \]

\[ \text{\emph{TellPos}} ::= \text{\emph{Head +: ConstraintBody}} \]
\[ \text{\emph{TellNeg}} ::= \text{\emph{Head -: ConstraintBody}} \]
\[ \text{\emph{AskPos}} ::= \text{\emph{Head +? Indexical}} \]
\[ \text{\emph{AskNeg}} ::= \text{\emph{Head -: Indexical}} \]
\[ \text{\emph{ConstraintDef}} ::= \text{\emph{TellPos}.} \cdot \text{?} (\text{\emph{TellNeg}}.)
\]
\[ \text{? (\text{\emph{AskPos}}.)} \]
\[ \text{? (\text{\emph{AskNeg}}.)} \]

34.11.2 Syntax of Arithmetic Expressions

\[ X ::= \text{\emph{variable}} \{ \text{domain variable} \} \]
\[ N ::= \text{\emph{integer}} \]
\[ \text{\emph{LinExpr}} ::= N \{ \text{linear expression} \} \]
\[ \mid X \]
\[ \mid N \times X \]
\[ \mid N \times N \]
\[ \mid \text{\emph{LinExpr}} + \text{\emph{LinExpr}} \]
\[ \mid \text{\emph{LinExpr}} - \text{\emph{LinExpr}} \]
\[ \text{\emph{Expr}} ::= \text{\emph{LinExpr}} \]
\[ \mid \text{\emph{Expr}} + \text{\emph{Expr}} \]
\[ \mid \text{\emph{Expr}} - \text{\emph{Expr}} \]
\[ \mid \text{\emph{Expr}} \times \text{\emph{Expr}} \]
\[ \mid \text{\emph{Expr}} / \text{\emph{Expr}} \{ \text{integer division} \} \]
\[ \mid \text{\emph{Expr}} \text{ mod } \text{\emph{Expr}} \]
\[ \mid \text{\emph{min}}(\text{\emph{Expr}}, \text{\emph{Expr}}) \]
\[ \mid \text{\emph{max}}(\text{\emph{Expr}}, \text{\emph{Expr}}) \]
\[ \mid \text{\emph{abs}}(\text{\emph{Expr}}) \]
\[ \text{\emph{RelOp}} ::= \# = \mid \# \neq \mid \# < \mid \# =< \mid \# > \mid \# >= \]
34.11.3 Operator Declarations

:- op(1200, xfx, [+,-,*,?,-?]).
:- op(760, yfx, #<=>).
:- op(750, xfy, #=>).
:- op(750, yfx, #<=).
:- op(740, yfx, #\/).
:- op(730, yfx, #\).
:- op(720, yfx, #\/).
:- op(710, fy, #\).
:- op(700, xfx, [in,in_set]).
:- op(700, xfx, [#=,#\=,#\<,#\<=,#\>,#\>=]).
:- op(550, xfx, ..).
:- op(500, fy, \).
:- op(490, yfx, ?).
:- op(400, yfx, [/>,/<]).