Compile once, run many times.

Does consulting your CHR program take a long time? Probably it takes the CHR compiler a long time to compile the CHR rules into Prolog code. When you disable optimizations the CHR compiler will be a lot quicker, but you may lose performance.

10.34 Constraint Logic Programming over Finite Domains—library(clpfd)

10.34.1 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 10.34.2 [CLPFD Interface], page 497. A description of all constraints that the solver provides is contained in Section 10.34.3 [Available Constraints], page 500. The predicates for searching for solution are documented in Section 10.34.4 [Enumeration Predicates], page 518. The predicates for getting execution statistics are documented in Section 10.34.5 [Statistics Predicates], page 521. A few example programs are given in Section 10.34.10 [Example Programs], page 537. Finally, Section 10.34.11 [Syntax Summary], page 540 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 10.34.6 [Answer Constraints], page 521. The solver’s execution mechanism and primitives are described in Section 10.34.7 [The Constraint System], page 522. How to add new global constraints via a programming interface is described in Section 10.34.8 [Defining Global Constraints], page 523. How to define new primitive constraints with indexicals is described in Section 10.34.9 [Defining Primitive Constraints], page 529.
10.34.1.1 Referencing this Software

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


10.34.1.2 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 with contributions by Nicolas Beldiceanu. Péter Szeredi contributed material for this manual chapter.

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

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

10.34.2 Solver Interface

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 smaller and smaller 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: if a term containing domain variables is written, copied, asserted, gathered as a solution to findall/3 and friends, or raised as an exception, those domain variables will be replaced by brand new variables in the copy. To retain the domain variables and any attached constraints, you can use copy_term/3 (see Section 4.8.7 [ref-lte-cpt], page 112).

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 Díaz’s clp(FD) implementation [Díaz & Codognet 93], one of the fastest finite domains solvers around.

10.34.2.1 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.

10.34.2.2 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 constraints 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).

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

mm([S,E,N,D,M,O,R,Y], Type) :-
    domain([S,E,N,D,M,O,R,Y], 0, 9), % step 1
    S#>0, M#>0,
    all_different([S,E,N,D,M,O,R,Y]), % step 2
    sum(S,E,N,D,M,O,R,Y),
    labeling(Type, [S,E,N,D,M,O,R,Y]). % step 3

sum(S, E, N, D, M, O, R, Y) :-
    1000*S + 100*E + 10*N + D
    + 1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y.
```

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

### 10.34.2.3 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 \textit{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:

\[
| ?- \text{Constraint} \#<=> B. 
\]

where \text{Constraint} is reifiable. As an example of a constraint that uses reification, consider \texttt{exactly}(\( X, L, N \)), which is true if \( X \) occurs exactly \( N \) times in the list \( L \). It can be defined thus:

\[
\text{exactly}(\_, [], 0).
\]
\[
\text{exactly}(X, [Y|L], N) :-
\]
\[
X \#= Y \#<=> B,
\]
\[
N \#= M+B,
\]
\[
\text{exactly}(X, L, M).
\]

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

10.34.3.1 Arithmetic Constraints

\texttt{?Expr RelOp ?Expr}

defines an arithmetic constraint. The syntax for \texttt{Expr} and \texttt{RelOp} is defined by a grammar (see Section 10.34.11.2 [Syntax of Arithmetic Expressions], page 542). 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 bound-consistency and their reified versions detect bound-entailment and -disentailment; see Section 10.34.7 [The Constraint System], page 522.

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.
Chapter 10: The Prolog Library 501

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 \( \text{sum}(Xs) \text{ RelOp Value} \). Cannot be reified. Corresponds roughly to sumlist/2 in library(lists).

scalar_product(+Coeffs, +Xs, +RelOp, ?Value)
scalar_product(+Coeffs, +Xs, +RelOp, ?Value, +Options)
where Coefs 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 \( \text{sum}(\text{Coeffs} \times Xs) \text{ RelOp Value} \). Cannot be reified.

Options is a list that may include the following option. It can be used to control the level of consistency used by the constraint.

consistency(Cons)
The value is one of the following:

domain The constraint will maintain arc-consistency. Please note: This option is only meaningful if RelOp is #=, and requires that any domain variables have finite bounds.

bound value The constraint will try to maintain bound-consistency (the default).

The following constraints constrain a value to be the minimum (maximum) of a given list of values.

minimum(?Value, +Xs)
where Xs is a list of integers or domain variables, and Value is an integer or a domain variable. True if Value is the minimum of Xs. Cannot be reified. Corresponds to min_member/2 in library(lists).

maximum(?Value, +Xs)
where Xs is a list of integers or domain variables, and Value is an integer or a domain variable. True if Value is the maximum of Xs. Cannot be reified. Corresponds to max_member/2 in library(lists).

10.34.3.2 Membership Constraints
domain(+Variables, +Min, +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 in +Range defines a membership constraint. X is an integer or a domain variable and Range is a ConstantRange (see Section 10.34.11.1 [Syntax of Indexicals], page 540). True if X is an element of the range.
?X in_set +FDSet

defines a membership constraint. X is an integer or a domain variable and FDSet is an FD set term (see Section 10.34.8.3 [FD Set Operations], page 526).

True if X is an element of the FD set.

in/2 and in_set/2 constraints can be reified. They maintain arc-consistency and their reified versions detect arc-entailment and -disentailment; see Section 10.34.7 [The Constraint System], page 522.

10.34.3.3 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 #\ Y #= 6

epresses 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 10.34.9 [Defining Primitive Constraints], page 529. The following propositional combinators are available:

#:Q

True if the constraint Q is false.

:#\ :P #\ :Q

True if the constraints P and Q are both true.

:#\ :P #\ :Q

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

:#\ / :P #\ / :Q

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

:#\ => :P

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

:#\ <= :Q

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

Note that the reification scheme introduced in Section 10.34.2.3 [Reified Constraints], page 499 is a special case of a propositional constraint.

10.34.3.4 Combinatorial Constraints

The constraints listed here are sometimes called symbolic constraints. They are currently not reifiable. Unless documented otherwise, they maintain (at most) bound-consistency in their arguments; see Section 10.34.7 [The Constraint System], page 522.
global_cardinality(+Xs,+Vals)
global_cardinality(+Xs,+Vals,+Options)
where Xs = [X1, ..., Xd] is a list of integers or domain variables, and Vals = [K1-V1, ..., Kn-Vn] is a list of pairs where each key Ki is a unique integer and Vi is a domain variable or an integer. True if every element of Xs is equal to some key and for each pair Ki-Vi, exactly Vi elements of Xs are equal to Ki.

If either Xs or Vals is ground, and in many other special cases, global_cardinality/[2,3] maintains arc-consistency, but generally, bound-consistency cannot be guaranteed. An arc-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:

consistency(Cons)
Which filtering algorithm to use. One of the following:

- domain: The constraint will use the algorithm mentioned above. Implies on(dom). The default.
- bound: The constraint will use the algorithm mentioned above. Implies on(minmax).
- value: The constraint will use a simple algorithm, which prevents too few or too many of the Xs from taking values among the Vals. Implies on(val).

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

- dom: to wake up when the domain of a variable is changed (the default);
- minmax: to wake up when the domain bound of a variable is changed;
- val: to wake up when a variable becomes ground.

cost(Cost,Matrix)
Overrides any consistency/1 option value. A cost is associated with the constraint and reflected into the domain variable Cost. Matrix should be a d*n matrix, represented as a list of d lists, each of length n. Assume that each Xi equals K(pi). The cost of the constraint is then Matrix[1,p1]+...+Matrix[d, pd].

With this option, an arc-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.

element(?X,+List,+Y)
where X and Y are integers or domain variables and List is a list of integers or domain variables. True if the X:th element of 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.

Maintains arc-consistency in X and bound-consistency in List and Y. Corresponds to nth1/3 in library(lists).
table(+Tuples,+Extension)

Defines an n-ary constraint by extension. \textit{Extension} should be a list of lists of integers, each of length \textit{n}. \textit{Tuples} should be a list of lists of domain variables or integers, each also of length \textit{n}. The constraint holds if every \textit{Tuple} in \textit{Tuples} occurs in the \textit{Extension}.

For convenience, \textit{Extension} may contain \textit{ConstantRange} (see Section 10.34.11.1 [Syntax of Indexicals], page 540) expressions in addition to integers.

\textit{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:

- \textbf{consistency(Cons)}
  - The value is one of the following:
    - \textit{domain} The constraint will maintain arc-consistency (the default).
    - \textit{bound} The constraint will maintain bound-consistency.
    - \textit{value} The constraint will wake up when a variable has become ground, and only prune a variables when its domain has been reduced to a singleton.

\texttt{table/[2,3]} is implemented in terms of the following, more general constraint, with which arbitrary relations can be defined compactly:

case(+Template, +Tuples, +Dag)

case(+Template, +Tuples, +Dag, +Options)

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

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

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

Nodes are either \textit{internal nodes} or \textit{leaf nodes}. In the former case, \textit{Successors} is a list of terms \texttt{(Min..Max)-ID2}, where the \texttt{ID2} refers to a child node. In the latter case, \textit{Successors} is a list of terms \texttt{(Min..Max)}. In both cases, the \texttt{(Min..Max)} should form disjoint intervals.

\textit{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 \textit{Template} should occur exactly once on each path, and there must not be any cycles.

\textit{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)

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

\textbf{on}(Spec) \quad \text{specifies how eagerly the constraint should react to domain changes of } X.

\textbf{prune}(Spec) \quad \text{specifies the extent to which the constraint should prune the domain of } X.

\textit{Spec} is one of the following, where \( X \) is a place-holder variable occurring in \textit{Template} or equal to \textit{TLeaf}:

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

\textbf{min}(X) \quad \text{wake up when the lower bound of } X \text{ has changed, resp. prune only the lower bound of } X.

\textbf{max}(X) \quad \text{wake up when the upper bound of } X \text{ has changed, resp. prune only the upper bound of } X.

\textbf{minmax}(X) \quad \text{wake up when the lower or upper bound of } X \text{ has changed, resp. prune only the bounds of } X.

\textbf{val}(X) \quad \text{wake up when } X \text{ has become ground, resp. only prune } X \text{ when its domain has been reduced to a singleton.}

\textbf{none}(X) \quad \text{ignore domain changes of } X, \text{ resp. never prune } X.

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

\textit{path}(ID, Tuple, Leaf) holds if \textit{Dag} contains a term \textit{node(ID, Var, Successors)}, \textit{Var} is the unique \( k \):th element of \textit{Template}, \( i \) is the \( k \):th element of \textit{Tuple}, and:

1. \textit{Successors} contains a term \((\textit{Min..Max})-\textit{Child})
2. \( \text{Min} \leq i \leq \text{Max} \), and
3. \textit{path}(Child, Tuple, Leaf) holds; or

The node is a leaf node, and

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

For example, recall that \textit{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:

\textit{element}(X, [1,1,1,1,2,2,2,2], Y),
\textit{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:

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

The DAG of the previous example has the following shape:

A couple of sample queries:

```
```

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 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:

consistency(Cons)
Which algorithm to use, one of the following:

domain The default for all_distinct/[1,2] and assignment/[2,3]. An arc-consistency algorithm [Regin 94] is used, roughly linear in the total size of the domains. Implies on(dom).

bound A bound-consistency algorithm [Mehlhorn 00] is used. This algorithm is nearly linear in the number of variables and values. Implies on(minmax).

value 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. Implies on(val).
on(On)  How eagerly to wake up the constraint. One of the following:
  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.

nvalue(?N, +Variables)
  where Variables is a list of domain variables with finite bounds or integers, and N is an integer or a domain variable. True if N is the number of distinct values taken by Variables. Approximates bound-consistency in N and arc-consistency in Variables. Can be thought of as a relaxed version of all_distinct/2.

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 in [1,n] and Xi=j iff Yj=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, an arc-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 so that the total resource consumption does not exceed a given limit at any time:

```
cumulative(+Tasks)
cumulative(+Tasks, +Options)
```

A task is represented by a term `task($Oi,Di,Ei,Hi,Ti$)` where $Oi$ is the start time, $Di$ the non-negative duration, $Ei$ the end time, $Hi$ the non-negative resource consumption, and $Ti$ the task identifier. All fields are domain variables with bounded domains, or integers.

Let $n$ be the number of tasks and $L$ the global resource limit (by default 1, but see below), and:

$$
H_{ij} = Hi, \text{ if } Oi =< j <= Oi + Di \\
H_{ij} = 0, \text{ otherwise}
$$

The constraint holds if:

1. For every task $i$, $Si + Di = Ei$, and
2. For all instants $j$, $H_{1j} + . . . + H_{nj} =< L$.

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

- `limit(L)` See above.
- `precedences(Ps)`
  `Ps` encodes a set of precedence constraints to apply to the tasks.
  `Ps` should be a list of terms of the form:

  $$
  Ti - Tj \#= Dij
  $$

  where $Ti$ and $Tj$ should be task identifiers, and $Dij$ should be a a domain variable (or an integer), denoting:

  $$
  Oi - Oj = Dij \text{ and } Dij \text{ in } r
  $$

- `global(Boolean)`
  if `true`, a more expensive algorithm will be used in order to achieve tighter pruning of the bounds of the parameters.

This constraint is due to Aggoun and Beldiceanu [Aggoun & Beldiceanu 93].

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.

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

A task is represented by a term \(\text{task}(O_i,D_i,E_i,H_i,M_i)\) where \(O_i\) is the start time, \(D_i\) the non-negative duration, \(E_i\) the end time, \(H_i\) the resource consumption (if positive) or production (if negative), and \(M_i\) a machine identifier. All fields are domain variables with bounded domains, or integers.

A machine is represented by a term \(\text{machine}(M_j,L_j)\) where \(M_j\) is the identifier and \(L_j\) is the resource bound of the machine. Both fields must be integers.

Let there be \(n\) tasks and:

\[
H_{ijm} = H_i, \text{ if } M_i=m \text{ and } O_i =< j < O_i+D_i
\]
\[
H_{ijm} = 0 \text{ otherwise}
\]

If the resource bound is \text{lower} (the default), the constraint holds if:
1. For every task \(i\), \(S_i+D_i=E_i\), and
2. For all machines \(m\) and instants \(j\) such that there exists a task \(i\) where \(M_i=m\) and \(O_i =< j < O_i+D_i\), \(H_{1jm}+\ldots+H_{njm} >= L_m\).

If the resource bound is \text{upper}, the constraint holds if:
1. For every task \(i\), \(S_i+D_i=E_i\), and
2. For all machines \(m\) and instants \(j\), \(H_{1jm}+\ldots+H_{njm} =< L_m\).

\text{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):

- \text{bound(B)}: If \text{lower} (the default), each resource limit is treated as a lower bound. If \text{upper}, each resource limit is treated as an upper bound.
- \text{prune(P)}: If \text{all} (the default), the constraint will try to prune as many variables as possible. If \text{next}, only variables that occur in the first non-ground task term (wrt. the order given when the constraint was posted) can be pruned.
- \text{generalization(Boolen)}
  - If \text{true}, extra reasoning based on assumptions on machine assignment will be done to infer more.
- \text{task_intervals(Boolen)}
  - If \text{true}, extra global reasoning will be performed in an attempt to infer more.

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.
P is a permutation of [1,n].
for all i in [1,n] : Xi = Y(P_i)

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

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

\[
\text{disjoint1(} + \text{Lines} \text{)} \\
\text{disjoint1(} + \text{Lines,} + \text{Options} \text{)}
\]

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):

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

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

\[
\text{wrap(} \text{Min,} \text{Max} \text{)}
\]

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

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

This option imposes a minimal distance \( D \) between the end point
of any line of type \( \text{T1} \) and the origin of any line of type \( \text{T2} \). \( D \)
should be a positive integer or sup. If sup is used, all lines of type
\( \text{T2} \) must be placed before any line of type \( \text{T1} \).
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
\text{disjoint1/2} is used for scheduling non-overlapping tasks.

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

where Rectangles is a list of terms \( F(X_j, L_j, Y_j, H_j) \) or \( F(X_j, L_j, Y_j, H_j, T_j) \), \( X_j \) and
\( L_j \) are domain variables with finite bounds or integers denoting the origin and
size of rectangle \( j \) in the X dimension, \( Y_j \) and \( H_j \) 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):

**decomposition** (**Boolean**)

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

**global** (**Boolean**)

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

**wrap** (**Min1**, **Max1**, **Min2**, **Max2**)

**Min1** and **Max1** should be either integers or the atoms \( \text{inf} \) and \( \text{sup} \) 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 \( \text{Min1} \) and \( \text{Max1} \) coincide. Furthermore, this option forces the domains of the \( X_j \) variables to be inside \([\text{Min1}, \text{Max1}-1]\).

\( \text{Min2} \) and \( \text{Max2} \) should be either integers or the atoms \( \text{inf} \) and \( \text{sup} \) 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 \( \text{Min2} \) and \( \text{Max2} \) coincide. Furthermore, this option forces the domains of the \( Y_j \) variables to be inside \([\text{Min2}, \text{Max2}-1]\).

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

**margin** (**T1**, **T2**, **D1**, **D2**)

This option imposes minimal distances \( D1 \) in the X dimension and \( D2 \) in the Y dimension between the end point of any rectangle of type \( T1 \) and the origin of any rectangle of type \( T2 \). \( D1 \) and \( D2 \) should be positive integers or \( \text{sup} \). If \( \text{sup} \) is used, all rectangles of type \( T2 \) must be placed before any rectangle of type \( T1 \) 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.

**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 \textit{Synch} is \textit{true}, it can be solved with first-fail labeling in 23 backtracks. If \textit{Synch} is \textit{false}, 60 million backtracks do not suffice to solve it.

\begin{verbatim}
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(O1,3,Y1a,1),
                t(O1,3,Y1b,1),
                t(O1,3,Y1c,1),
                t(O2,5,Y2a,1),
                t(O2,5,Y2b,1),
                t(O2,5,Y2c,1),
                t(O2,5,Y2d,1),
                t(O3,9,Y3a,1),
                t(O3,9,Y3b,1),
                t(O3,9,Y3c,1),
                t(O3,9,Y3d,1),
                t(O4,6,Y4a,1),
                t(O4,6,Y4b,1),
                t(O4,6,Y4c,1)],
   [synchronization(Synch)]).
\end{verbatim}

The file \texttt{library('clpfd/examples/squares.pl')} contains an example where \texttt{disjoint2/2} is used for tiling squares.

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(0p)  
  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.

- increasing 
  This option imposes the additional constraint that each vector in Vectors be sorted in strictly ascending order.

- among(Least,Most,Values) 
  If given, Least and Most should be integers such that 0 =< Least =< 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 belong to Values.

Unless the increasing/0 or among/3 options are given, the underlying algorithm [Carlsson & Beldiceanu 02] guarantees arc-consistency.

The following constraint provides a general way of defining any constraint involving sequences whose checker, i.e. a procedure that classifies ground instances as solutions or non-solutions, can be expressed by a finite automaton, extended with counter operations on its arcs. The point is that it is very much easier to come up with such a checker that to come up with a filtering algorithm for the constraint of interest.

automaton(...)

The constraint has the form ctr according to the grammar shown below, which describes its abstract syntax. The arguments are:

- sequence The sequence of terms of interest.
- template A template for an item of the sequence. Only relevant if some state transition involving counter arithmetic mentions a variable occurring in template, in which case the corresponding term in a sequence element will be accessed.
- signature The signature of sequence. The automaton is not driven by the sequence itself, but by signature, which ranges over an alphabet, defined in the following argument. In addition to automaton/8, you must call a constraint that maps sequence to signature.
- nodes The nodes of the automaton, classified as source, sink or internal.
- arcs The arcs (transitions) of the automaton. Any transition not mentioned is assumed to go to an implicit failure node. An arc optionally contains expressions for updated counter values; by default, the counters remain unchanged. Conditional updates can be specified.
counters For $k$ counters, a list of $k$ variables.
initial For $k$ counters, a list of $k$ initial values, usually instantiated.
nal For $k$ counters, a list of $k$ final values, usually uninstantiated.

Abstract syntax:

```
ctr ::= automaton(sequence, template, signature, nodes, arcs, counters, initial, final)
sequence ::= list of template {all of which of the same shape}
template ::= term {most general shape of the sequence}
{its variables should be local to ctr}
signature ::= list of variable
nodes ::= list of nodespec {all of which of the same shape}
nodespec ::= source(node) {the initial state}
| sink(node) {an accept state}
node ::= atomic
arcs ::= list of arc {all of which of the same shape}
ar ::= arc(node, integer, node) {from node, integer, to node}
| arc(node, integer, node, exprs) {exprs correspond to new counter values}
| arc(node, integer, node, conditional)
conditional ::= (cond -> exprs) {of same length as counters}
| (conditional ; conditional)
exprs ::= list of Expr
{Expr as defined in Section 10.34.11.2 [Syntax of Arithmetic Expressions], page 542}
{over counters, template and constants}
{variables occurring in counters correspond to old counter values}
{variables occurring in template refer to the current element of sequence}
cond ::= constraint {over counters, template and constants}
{must be reifiable or true}
counters ::= list of variable {should be local to ctr}
initial ::= list of integer {of same length as counters}
nal ::= list of variable {of same length as counters}
```
If no counters are used, the arguments `counters`, `initial` and `final` should be `[]`. The arguments `template` and `sequence` are only relevant if some `Expr` mentions a variable in `template`, in which case the corresponding position in `sequence` will be used at that point.

The constraint holds for a ground instance `sequence` if:

- `signature` is the signature corresponding to `sequence`.
- The finite automaton encoded by `nodes` and `arcs` stops in an accept state.
- Any counter arithmetic on the transitions map their `initial` values to the `final` values.

Here is an example. Suppose that you want to define the predicate `inflexion(N, L)` which should hold if `L` is a list of domain variables, and `N` is the number of times that the sequence order switches between strictly increasing and strictly decreasing. For example, the sequence `[1,1,4,8,8,2,7,1]` switches order three times.

Such a constraint is conveniently expressed by a finite automaton over the alphabet `[/,<,=,>]` denoting the order between consecutive list elements. A counter is incremented when the order switches, and is mapped to the first argument of the constraint. The automaton could look as follows:

![Automaton for inflexion/2](image)

The following piece of code encodes this using `automaton/8`. The auxiliary predicate `inflexion_signature/2` maps the sequence to a signature where the consecutive element order is encoded over the alphabet `[0,1,2]`. We use one counter with initial value 0 and final value `N` (an argument of `inflexion/2`). Two transitions increment the counter. All states are accept states.
inflexion(N, VARIABLES) :-
inflexion_signature(VARIABLES, SIGNATURE),
automaton(_, _, SIGNATURE,
  [source(s),sink(i),sink(j),sink(s)],
  [arc(s,1,s ),
   arc(s,2,i ),
   arc(s,0,j ),
   arc(i,1,i ),
   arc(i,2,i ),
   arc(i,0,j,[C+1]),
   arc(j,1,j ),
   arc(j,0,j ),
   arc(j,2,i,[C+1])],
  [C],[0],[N]).

inflexion_signature([], []).
inflexion_signature([], []) :- !.
inflexion_signature([VAR1,VAR2|VARs], [S|Ss]) :-
  S in 0..2,
  VAR1 #> VAR2 #=> S #= 0,
  VAR1 #= VAR2 #=> S #= 1,
  VAR1 #< VAR2 #=> S #= 2,
  inflexion_signature([VAR2|VARs], Ss).

A couple of queries:

| ?- inflexion(N, [1,1,4,8,8,2,7,1]).
  N = 3 ? RET
  yes

| ?- length(L, 4), domain(L, 0, 1), inflexion(2,L), labeling([],L).
  L = [0,1,0,1] ? ;
  L = [1,0,1,0] ? ;
  no

This constraint was introduced in [Beldiceanu, Carlsson & Petit 04].

10.34.3.5 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 10.34.8 [Defining Global Constraints], page 523. 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 predi-
cates; see Section 10.34.9 [Defining Primitive Constraints], page 529. They can optionally
be reified.
10.34.4 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:

\[ \text{indomain}(\, ?X \, ) \]
where \( X \) is a domain variable with a bounded domain or an integer. Assigns, in increasing order via backtracking, a feasible value to \( X \).

\[ \text{labeling}(\, :\text{Options}, \, +\text{Variables} \, ) \]
where \( \text{Variables} \) is a list of domain variables or integers and \( \text{Options} \) 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.

\[ \text{first_bound}(\, +\text{BB0}, \, -\text{BB} \, ) \]
\[ \text{later_bound}(\, +\text{BB0}, \, -\text{BB} \, ) \]
Provides an auxiliary service for the \text{value} (\text{Enum}) option (see below).

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

The \text{Options} argument of \text{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.

\text{leftmost} \quad \text{The leftmost variable is selected. This is the default.}
\text{min} \quad \text{The leftmost variable with the smallest lower bound is selected.}
\text{max} \quad \text{The leftmost variable with the greatest upper bound is selected.}
\text{ff} \quad \text{The first-fail principle is used: the leftmost variable with the smallest domain is selected.}
\text{ffc} \quad \text{The most constrained heuristic is used: a variable with the smallest domain is selected, breaking ties by (a) selecting the variable...}
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 \( X #= B \) and \( X \#\neq B \), where \( B \) is the lower or upper bound of \( X \). This is the default.

denum Makes a multiple choice for \( X \) corresponding to the values in its domain.

bisect Makes a binary choice between \( X \#=< M \) and \( X \#> M \), where \( M \) is the midpoint of the domain of \( X \). This strategy is also known as domain splitting.

value(Enum)

Enum is a predicate that should prune the domain of \( X \), possibly but not necessarily to a singleton. \( \) It will be called as Enum(X,Rest,BB0,BB) where 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, pruning the domain of \( X \), and to backtrack one or more times, providing alternative prunings. 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,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.

\begin{itemize}
  \item \texttt{all}  All solutions are enumerated. This is the default.
  \item \texttt{minimize(} \( X \) \texttt{)}  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 \texttt{Variables}. It is useful to combine these option with the \texttt{time_out/2} option (see below). If these options occur more than once, the last occurrence overrides previous ones.
  \item \texttt{maximize(} \( X \) \texttt{)}
\end{itemize}

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

\texttt{assumptions(} \( K \) \texttt{)}  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:

\texttt{discrepancy(} \( D \) \texttt{)}  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:

\texttt{time_out(} \( Time, \) \texttt{Flag) } This is equivalent to a goal \texttt{time_out(labeling(...),Time,Flag)} (see Section 10.25 [lib-timeout], page 472). Furthermore, if combined with the \texttt{minimize(V)} or \texttt{maximize(V)} option, and the time limit is reached, the values of \texttt{Variables} and \( V \) will be those of the best solution found.

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

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

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:

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

The file \texttt{library(clpfd/examples/tsp.pl)} contains an example of user-defined variable and value choice heuristics.
10.34.5 Statistics Predicates

The following predicates can be used to get execution statistics.

\textbf{fd_statistics(?\text{Key}, ?\text{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 \texttt{statistics/2}.

For each of the possible keys \texttt{Key}, \texttt{Value} is unified with the current value of a counter, which is simultaneously zeroed. The following counters are maintained. See Section 10.34.7 [The Constraint System], page 522, for details of what they all mean:

- \texttt{resumptions} The number of times a constraint was resumed.
- \texttt{entailments} The number of times a (dis)entailment was detected by a constraint.
- \texttt{prunings} The number of times a domain was pruned.
- \texttt{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.
- \texttt{constraints} The number of constraints created.

\texttt{fd_statistics}

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

10.34.6 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:

\texttt{clpfd:full_answer}

\texttt{hook}

If false (the default), the answer constraint, as well as constraints projected by \texttt{clpfd:project_attributes/2}, \texttt{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.
10.34.7 The Constraint System

10.34.7.1 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 *arc-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 *arc-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 *bound-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 *bound-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 *arc-disentailed (bound-disentailed) by $S$* iff its negation is arc-entailed (bound-entailed) by $S$.

10.34.7.2 Pitfalls of Interval Reasoning

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

| ?- X+Y #= Z, X=1, Z=6, Y in 1..10, Y #\= 5. |
| no |
| ?- X+Y #= Z #=> B, X=1, Z=6, Y in 1..10, Y #\= 5. |
| X = 1, |
| Z = 6, |
| Y in(1..4)\/(6..10), |
| B in 0..1 |
Since $1+5#6$ holds, $X+Y # Z$ is not bound-disentailed, although any attempt to make it bound-consistent wrt. the store results in a contradictory store.

### 10.34.8 Defining Global Constraints

#### 10.34.8.1 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:

```prolog
clpfd:dispatch_global(+Constraint, +State0, -State, -Actions) extendible
```

Tells the solver how to solve constraints of the form `Constraint`. Defined as a 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 `clpfd:dispatch_global/4`.

`State0` and `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, `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 $X$ to $V$.
- $X$ in $R$ The solver constrains $X$ to be a member of the `ConstantRange R` (see Section 10.34.11.1 [Syntax of Indexicals], page 540).
- $X$ in_set $S$ The solver constrains $X$ to be a member of the FD set $S$ (see Section 10.34.8.3 [FD Set Operations], page 526).
call(\textit{Goal})

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

\textit{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.

\begin{verbatim}
fd_global(:\textit{Constraint}, +\textit{State}, +\textit{Susp})
fd_global(:\textit{Constraint}, +\textit{State}, +\textit{Susp}, +\textit{Options})
\end{verbatim}

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

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

\begin{itemize}
  \item \texttt{dom(X)} \quad wake up when the domain of $X$ has changed
  \item \texttt{min(X)} \quad wake up when the lower bound of $X$ has changed
  \item \texttt{max(X)} \quad wake up when the upper bound of $X$ has changed
  \item \texttt{minmax(X)} \quad wake up when the lower or upper of $X$ has changed
  \item \texttt{val(X)} \quad wake up when $X$ has become ground
\end{itemize}

\textit{Options} is a list of zero or more of the following:

\begin{itemize}
  \item \texttt{source(\textit{Term})}
    \begin{itemize}
      \item By default, the symbolic form computed by \texttt{copy_term/3}, and
        shown in the answer constraint if \texttt{clpfd:full_answer} holds, equals
        \textit{Constraint}, module name expanded. With this option, the symbolic
        form will instead be \textit{Term}. In particular, if \textit{Term} equals \texttt{true},
        the constraint will not appear in the \texttt{Body} argument of \texttt{copy_term/3}.
        This can be useful if you are posting some redundant (implied)
        constraint.
    \end{itemize}
  \item \texttt{idempotent(\texttt{Boolean})}
    \begin{itemize}
      \item If \texttt{true} (the default), the constraint solving method is assumed
          to be idempotent. That is, in the scope of \texttt{clpfd:dispatch_}
          \texttt{global/4}, the solver will not check for the resumption conditions
          for the given constraint, while performing its \texttt{Actions}. If \texttt{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.
    \end{itemize}
\end{itemize}

For an example of usage, see Section 10.34.8.4 [A Global Constraint Example], page 528.

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).

### 10.34.8.2 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 10.34.11.1 [Syntax of Indexicals], page 540) denoting the 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).

\texttt{fd_neighbors(+Var, -Neighbors)}

Given a domain variable \texttt{Var}, \texttt{Neighbors} is the set of variables that can be reached from \texttt{Var} via constraints posted so far.

\texttt{fd_closure(+Vars, -Closure)}

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

10.34.8.3 FD Set Operations

The domains of variables are internally represented compactly as \textit{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:

\texttt{is_fdset(+Set)}

\texttt{Set} is a valid FD set.

\texttt{empty_fdset(?Set)}

\texttt{Set} is the empty FD set.

\texttt{fdset_parts(?Set, ?Min, ?Max, ?Rest)}

\texttt{Set} is an FD set, which is a union of the non-empty interval \([\texttt{Min}, \texttt{Max}]\) and the FD set \texttt{Rest}, and all elements of \texttt{Rest} are greater than \texttt{Max}+1. \texttt{Min} and \texttt{Max} are both integers or the atoms \texttt{inf} and \texttt{sup}, denoting minus and plus infinity, respectively. Either \texttt{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:

\texttt{empty_interval(+Min, +Max)}

\texttt{[Min, Max]} is an empty interval.

\texttt{fdset_interval(?Set, ?Min, ?Max)}

\texttt{Set} is an FD set, which is the non-empty interval \([\texttt{Min}, \texttt{Max}]\).

\texttt{fdset_singleton(?Set, ?Elt)}

\texttt{Set} is an FD set containing \texttt{Elt} only. At least one of the arguments must be ground.

\texttt{fdset_min(+Set, -Min)}

\texttt{Min} is the lower bound of \texttt{Set}.

\texttt{fdset_max(+Set, -Min)}

\texttt{Max} is the upper bound of \texttt{Set}. This operation is linear in the number of intervals of \texttt{Set}. 

\texttt{is_fdset(+Set)}

\texttt{Set} is a valid FD set.

\texttt{empty_fdset(?Set)}

\texttt{Set} is the empty FD set.

\texttt{fdset_parts(?Set, ?Min, ?Max, ?Rest)}

\texttt{Set} is an FD set, which is a union of the non-empty interval \([\texttt{Min}, \texttt{Max}]\) and the FD set \texttt{Rest}, and all elements of \texttt{Rest} are greater than \texttt{Max}+1. \texttt{Min} and \texttt{Max} are both integers or the atoms \texttt{inf} and \texttt{sup}, denoting minus and plus infinity, respectively. Either \texttt{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:

\texttt{empty_interval(+Min, +Max)}

\texttt{[Min, Max]} is an empty interval.

\texttt{fdset_interval(?Set, ?Min, ?Max)}

\texttt{Set} is an FD set, which is the non-empty interval \([\texttt{Min}, \texttt{Max}]\).

\texttt{fdset_singleton(?Set, ?Elt)}

\texttt{Set} is an FD set containing \texttt{Elt} only. At least one of the arguments must be ground.

\texttt{fdset_min(+Set, -Min)}

\texttt{Min} is the lower bound of \texttt{Set}.

\texttt{fdset_max(+Set, -Min)}

\texttt{Max} is the upper bound of \texttt{Set}. This operation is linear in the number of intervals of \texttt{Set}. 

Chapter 10: The Prolog Library

\[ \text{fdset\_size(+Set, -Size)} \]
Size is the cardinality of Set, represented as sup if Set is infinite. This operation is linear in the number of intervals of Set.

\[ \text{list\_to\_fdset(+List, -Set)} \]
Set is the FD set containing the elements of List. Slightly more efficient if List is ordered.

\[ \text{fdset\_to\_list(+Set, -List)} \]
List is an ordered list of the elements of Set, which must be finite.

\[ \text{range\_to\_fdset(+Range, -Set)} \]
Set is the FD set containing the elements of the ConstantRange (see Section 10.34.11.1 [Syntax of Indexicals], page 540) Range.

\[ \text{fdset\_to\_range(+Set, -Range)} \]
Range is a constant interval, a singleton constant set, or a union of such, denoting the same set as Set.

\[ \text{fdset\_add\_element(+Set1, +Elt, -Set2)} \]
Set2 is Set1 with Elt inserted in it.

\[ \text{fdset\_del\_element(+Set1, +Elt, -Set2)} \]
Set2 is like Set1 but with Elt removed.

\[ \text{fdset\_disjoint(+Set1, +Set2)} \]
The two FD sets have no elements in common.

\[ \text{fdset\_intersect(+Set1, +Set2)} \]
The two FD sets have at least one element in common.

\[ \text{fdset\_intersection(+Set1, +Set2, -Intersection)} \]
Intersection is the intersection between Set1 and Set2.

\[ \text{fdset\_intersection(+Sets, -Intersection)} \]
Intersection is the intersection of all the sets in Sets.

\[ \text{fdset\_member(?Elt, +Set)} \]
is true when Elt is a member of Set. If Elt is unbound, Set must be finite.

\[ \text{fdset\_eq(+Set1, +Set2)} \]
is true when the two arguments represent the same set i.e. they are identical.

\[ \text{fdset\_subset(+Set1, +Set2)} \]
Every element of Set1 appears in Set2.

\[ \text{fdset\_subtract(+Set1, +Set2, -Difference)} \]
Difference contains all and only the elements of Set1 that are not also in Set2.

\[ \text{fdset\_union(+Set1, +Set2, -Union)} \]
Union is the union of Set1 and Set2.

\[ \text{fdset\_union(+Sets, -Union)} \]
Union is the union of all the sets in Sets.

\[ \text{fdset\_complement(+Set, -Complement)} \]
Complement is the complement of Set wrt. inf..sup.
10.34.8.4 A Global Constraint Example

The following example defines a new global constraint \( \text{exactly}(X, L, N) \), 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 10.34.2.3 [Reified Constraints], page 499.

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
exactly(I, Xs, N) :-
  dom_suspensions(Xs, Susp),
  fd_global(exactly(I,Xs,N), state(Xs,N), Susp).

% the solver method
:- multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(exactly(I,_,_), state(Xs,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]
 ).
% 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([], _, []).
eq_all([X|Xs], Set, [X in_set Set|Ps]) :-
    eq_all(Xs, Set, Ps).
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

10.34.9 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.

10.34.9.1 Indexicals

An indexical is a reactive functional rule of the form \( X \in R \), where \( R \) is a set valued range expression (see below). See Section 10.34.11.1 [Syntax of Indexicals], page 540, 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.

10.34.9.2 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.

\[
\text{dom}(X) \quad \text{evaluates to } D(X,S) \\
\{T_1, \ldots, T_n\} \quad \text{evaluates to } \{S(T_1), \ldots, S(T_n)\}. \text{ Any } T_i \text{ containing a variable that is not “quantified” by unionof/3 will cause the indexical to suspend until this variable has been assigned.} \\
T_1 \ldots T_2 \quad \text{evaluates to } S(T_1) \text{ and } S(T_2). \\
R_1 \backslash R_2 \quad \text{evaluates to the intersection of } S(R_1) \text{ and } S(R_2) \\
R_1 \lor R_2 \quad \text{evaluates to the union of } S(R_1) \text{ and } S(R_2) \\
\backslash R_2 \quad \text{evaluates to the complement of } S(R_2) \\
R_1 \cdot R_2 \quad \text{evaluates to } S(R_2) \text{ or } S(T_2) \text{ added pointwise to } S(R_1) \\
- R_2 \quad \text{evaluates to } S(R_2) \text{ negated pointwise} \\
R_1 - R_2 \quad \text{evaluates to } S(R_2) \text{ or } S(T_2) \text{ subtracted pointwise from } S(R_1) \text{ or } S(T_1) \\
R_1 \mod R_2 \quad \text{evaluates to the pointwise modulo of } S(R_1) \text{ and } S(R_2) \text{ or } S(T_2) \\
R_1 \rem R_2 \quad \text{evaluates to the pointwise remainder of } S(R_1) \text{ and } S(R_2) \text{ or } S(T_2) \\
R_1 \oplus R_2 \quad \text{evaluates to } S(R_2) \text{ if } S(R_1) \text{ is a non-empty set; otherwise, evaluates to the empty set. This expression is commonly used in the context } (R_1 \oplus \text{inf..sup}) \lor R_3, \text{ which evaluates to } S(R_3) \text{ if } S(R_1) \text{ is an empty set; otherwise, evaluates} \\
\]
to \( \text{inf} \ldots \text{sup} \). As an optimization, \( R3 \) is not evaluated while the value of \( R1 \) is a non-empty set.

\[
\text{unionof}(X, R1, R2)
\]
evaluates to the union of \( S(E1), \ldots, S(EN) \), where each \( EI \) has been formed by substituting \( K \) for \( X \) in \( R2 \), where \( K \) is the \( I \)th element of \( S(R1) \). See Section 10.34.10.2 [N Queens], page 537, for an example of usage.

**Please note:** if \( S(R1) \) is infinite, the evaluation of the indexical will be abandoned, and the indexical will simply suspend.

\[
\text{switch}(T, MapList)
\]
evaluates to \( S(E) \) if \( S(T1) \) equals \( K \) and \( MapList \) 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.

### 10.34.9.3 Term Expressions

A term expression has one of the following forms, where \( T1 \) and \( T2 \) 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
- \( \neg T1 \) evaluates to \( S(T1) \) negated
- \( T1 + T2 \) evaluates to the sum of \( S(T1) \) and \( S(T2) \)
- \( T1 - T2 \) evaluates to the difference of \( S(T1) \) and \( S(T2) \)
- \( T1 \times T2 \) evaluates to the product of \( S(T1) \) and \( S(T2) \), where \( S(T2) \) must not be negative
- \( T1 \div T2 \) evaluates to the quotient of \( S(T1) \) and \( S(T2) \), rounded up, where \( S(T2) \) must be positive
- \( T1 \mod T2 \) evaluates to the modulo of \( S(T1) \) and \( S(T2) \)
- \( T1 \text{ rem } T2 \) evaluates to the remainder of \( S(T1) \) and \( S(T2) \)
10.34.9.4 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 \( \text{X in R} \) is (anti-)monotone iff \( R \) is (anti-)monotone.

The consistency or entailment of a constraint \( C \) expressed as indexicals \( \text{X in R} \) 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 10.34.9.6 [Execution of Propagating Indexicals], page 534 and Section 10.34.9.7 [Execution of Checking Indexicals], page 535.

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.

10.34.9.5 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:

\[
\text{plus}(X,Y,T) +: \\
\text{X in min(T) - max(Y) .. max(T) - min(Y)}, \\
\text{Y in min(T) - max(X) .. max(T) - min(X)}, \\
\text{T in min(X) + min(Y) .. max(X) + max(Y)}. \\
\]

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 bound-consistency, which is significantly cheaper to maintain than arc-consistency and suffices in most cases. The constraint could for example be used as follows:

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

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 smaller, 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 \(\text{plusd}\) below encodes the same \(X+Y=T\) constraint as the \(\text{plus}\) predicate above, but maintaining arc-consistency:

\[
\text{plusd}(X,Y,T) +:
X \text{ in } \text{dom}(T) - \text{dom}(Y),
Y \text{ in } \text{dom}(T) - \text{dom}(X),
T \text{ in } \text{dom}(X) + \text{dom}(Y).
\]

\[
\text{\textcolor{red}{| ?- \text{plusd}(X, Y, T).}}
\]

\[
X \text{ in } \{1\} \setminus \{3\},
Y \text{ in } \{10\} \setminus \{20\},
T \text{ in } \{11\} \setminus \{13\} \setminus \{21\} \setminus \{23\}
\]

This costs more in terms of execution time, but gives more precise results. For singleton domains \(\text{plus}\) and \(\text{plusd}\) 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, \textbf{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:

```prolog
'x\=y'(X,Y) +:
   X in \{Y\},
   Y in \{X\}.

'x\=y'(X,Y) -:
   X in dom(Y),
   Y in dom(X).

'x\=y'(X,Y) +?
   X in \dom(Y).

'x\=y'(X,Y) -?
   X in \{Y\}.
```

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

### 10.34.9.6 Execution of Propagating Indexicals

Consider the definition of a constraint $C$ containing a propagating indexical $X \in R$. 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 $X \in R$ 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

### 10.34.9.7 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 $\max(Y)$ in $R$ has been decreased

10.34.9.8 Goal Expanded Constraints

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

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 implementation by compilation to indexicals of the same constraints is also provided.

The following two constructions are semantically equivalent:

\[
\text{Head} +: \text{LinExpr} \text{ RelOp} \text{ LinExpr}.
\]

\[
\text{Head} :- \text{LinExpr} \text{ RelOp} \text{ LinExpr}.
\]

where $\text{Head}$ only contains unique variables mentioned in the linear arithmetic expressions $\text{LinExpr}$ (see Section 10.34.11.1 [Syntax of Indexicals], page 540). The alternative version of $\text{sum/8}$ in Section 10.34.10.1 [Send More Money], page 537 illustrates this technique.

Similarly, the following two constructions are semantically equivalent:

\[
\text{Pred}(X,Y) +: \text{element}(X, \text{CList}, Y).
\]

\[
\text{Pred}(X,Y) :- \text{element}(X, \text{CList}, Y).
\]

where $\text{CList}$ is a ground list of integers. This technique is used in some demo programs in library('clpfd/examples'). Please note: the generated indexical will assume that the domains of $X$ and $Y$ do not contain values incompatible with $\text{CList}$.

Similarly, the following two constructions are semantically equivalent:

\[
\text{Pred}(X_1,\ldots,X_n) +: \text{table}(\text{CTable}).
\]

\[
\text{Pred}(X_1,\ldots,X_n) :- \text{table}([X_1,\ldots,X_n], \text{CTable}).
\]

where $\text{CTable}$ is an $n$-ary relation given by extension, as for the constraint $\text{table/[2,3]}$. Please note: the generated indexical will assume that the domains of $X_1,\ldots,X_n$ do not contain values incompatible with $\text{CTable}$.

In the body of an FD predicate, $\text{element/3}$ and $\text{table/1}$ expressions expand to indexicals recursively built up from $\text{switch/2}$ and $\text{unionof/3}$ expressions. For example, the following constraint:

\[
p(X,Y) +: \text{table}([[1,1],[2,1..2],[3,1..3]]).
\]
expands to:

\[
q(X, Y) +: \quad \\
X \text{ in } \text{unionof}(B, \text{dom}(Y), \text{switch}(B, [1\rightarrow\{1,2,3\}, 2\rightarrow\{2,3\}, 3\rightarrow\{3\}))), \\
Y \text{ in } \text{unionof}(B, \text{dom}(X), \text{switch}(B, [1\rightarrow\{1\}, 2\rightarrow\{1,2\}, 3\rightarrow\{1,2,3\}])).
\]

### 10.34.10 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.
```

#### 10.34.10.1 Send More Money

Let us return briefly to the Send More Money problem (see Section 10.34.2.2 [A Constraint Satisfaction Problem], page 498). Its `sum/8` predicate will expand to a space-efficient conjunction 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:

\[
\text{sum}(S, E, N, D, M, O, R, Y) +: \\
1000*S + 100*E + 10*N + D \\
+ 1000*M + 100*O + 10*R + E \\
#= 10000*M + 1000*O + 100*N + 10*E + Y.
\]

#### 10.34.10.2 N Queens

The problem is to place N queens on an N×N 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

\[
\text{no_threat}(X, Y, I) :- \\
X \not\equiv Y, \\
X*+I \not\equiv Y, \\
X-I \not\equiv Y.
\]

However, this formulation introduces new temporary domain variables and creates twelve fine-grained indexicals. Worse, the disequalities only maintain bound-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 $2..3$, 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 \texttt{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 \texttt{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 \(\{Y\} \cup \{Y+I\} \cup \{Y-I\}\),
  Y in \(\{X\} \cup \{X+I\} \cup \{X-I\}\).

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

% version 3: strong but somewhat inefficient version
no_threat(X, Y, I) +:
  X in \((4..\text{card}(Y)) \cup (\text{inf}..\text{sup}) \cup \\text{unionof}(B, \text{dom}(Y), \(\{B\} \cup \{B+I\} \cup \{B-I\}\)),
  Y in \((4..\text{card}(X)) \cup (\text{inf}..\text{sup}) \cup \\text{unionof}(B, \text{dom}(X), \(\{B\} \cup \{B+I\} \cup \{B-I\}\)).*/

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

10.34.10.3 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:
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/2 constraint. Branch-and-bound search is used to find the minimal completion time.

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

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

schedule(Ss, End) :-
    Ss = [S1,S2,S3,S4,S5,S6,S7],
    Es = [E1,E2,E3,E4,E5,E6,E7],
    Tasks = [task(S1,16,E1, 2,0),
              task(S2, 6,E2, 9,0),
              task(S3,13,E3, 3,0),
              task(S4, 7,E4, 7,0),
              task(S5, 5,E5,10,0),
              task(S6,18,E6, 1,0),
              task(S7, 4,E7,11,0)],
    domain(Ss, 1, 30),
    domain(Es, 1, 50),
    domain([End], 1, 50),
    maximum(End, Es),
    cumulative(Tasks, [limit(13)]),
    append(Ss, [End], Vars),
    labeling([minimize(End)], Vars). % label End last
```

10.34.11 Syntax Summary
10.34.11.1 Syntax of Indexicals

\[ X ::= \text{variable} \quad \{ \text{domain variable} \} \]

\[ \text{Constant} ::= \text{integer} \]

\[ | \text{inf} \quad \{ \text{minus infinity} \} \]
\text{Term} ::= \text{Constant} \quad \{ \text{ plus infinity } \} \\
| \ X \quad \{ \text{ suspend until assigned } \} \\
| \ \min(X) \quad \{ \text{ min. of domain of X } \} \\
| \ \max(X) \quad \{ \text{ max. of domain of X } \} \\
| \ \text{card}(X) \quad \{ \text{ size of domain of X } \} \\
| \ - \ \text{Term} \\
| \ \text{Term} + \ \text{Term} \\
| \ \text{Term} - \ \text{Term} \\
| \ \text{Term} \ast \ \text{Term} \\
| \ \text{Term} \gg \ \text{Term} \quad \{ \text{ division rounded up } \} \\
| \ \text{Term} \ll \ \text{Term} \quad \{ \text{ division rounded down } \} \\
| \ \text{Term} \ \mod \ \text{Term} \\
| \ \text{Term} \ \rem \ \text{Term} \\
\text{TermSet} ::= \{ \text{Term}, \ldots, \text{Term} \} \\
\text{Range} ::= \text{TermSet} \\
| \ \text{dom}(X) \quad \{ \text{ domain of X } \} \\
| \ \text{Term} \ldots \ \text{Term} \quad \{ \text{ interval } \} \\
| \ \text{Range} \ \setminus \ \text{Range} \quad \{ \text{ intersection } \} \\
| \ \text{Range} \ \setminus \ \text{Range} \quad \{ \ \text{union} \} \\
| \ \setminus \ \text{Range} \quad \{ \ \text{complement} \} \\
| \ - \ \text{Range} \quad \{ \ \text{pointwise negation} \} \\
| \ \text{Range} + \ \text{Range} \quad \{ \ \text{pointwise addition} \} \\
| \ \text{Range} - \ \text{Range} \quad \{ \ \text{pointwise subtraction} \} \\
| \ \text{Range} \ \mod \ \text{Range} \quad \{ \ \text{pointwise modulo} \} \\
| \ \text{Range} \ \rem \ \text{Range} \quad \{ \ \text{pointwise remainder} \} \\
| \ \text{Range} + \ \text{Term} \quad \{ \ \text{pointwise addition} \} \\
| \ \text{Range} - \ \text{Term} \quad \{ \ \text{pointwise subtraction} \} \\
| \ \text{Term} - \ \text{Range} \quad \{ \ \text{pointwise subtraction} \} \\
| \ \text{Range} \ \mod \ \text{Term} \quad \{ \ \text{pointwise modulo} \} \\
| \ \text{Range} \ \rem \ \text{Term} \quad \{ \ \text{pointwise remainder} \} \\
| \ \text{Range} \ \cap \ \text{Range} \quad \{ \ \text{intersection} \} \\
| \ \text{Range} \ \cup \ \text{Range} \quad \{ \ \text{union} \} \\
| \ \setminus \ \text{Range} \quad \{ \ \text{complement} \} \\
| \ \text{Range} \ \setminus \ \text{Range} \quad \{ \ \text{intersection} \} \\
| \ \setminus \ \text{Range} \quad \{ \ \text{complement} \} \\
| \ \text{Range} \ \setminus \ \text{Range} \quad \{ \ \text{intersection} \} \\
\text{ConstantSet} ::= \{ \text{integer}, \ldots, \text{integer} \} \\
\text{ConstantRange} ::= \text{ConstantSet} \\
| \ \text{Constant} \ldots \ \text{Constant} \\
| \ \text{ConstantRange} \ \cap \ \text{ConstantRange} \\
| \ \text{ConstantRange} \ \cup \ \text{ConstantRange} \\
| \ \setminus \ \text{ConstantRange} \\
\text{MapList} ::= [] \quad \{ \text{integer} - \text{ConstantRange} | \text{MapList} \} \\
\text{CTable} ::= []
| [CRow | CTable ]

CRow  ::= []
| [integer | CRow ]
| [ConstantRange | CRow ]

CList  ::= []
| [integer | CList ]

Indexical ::= X in Range
Indexicals ::= Indexical
| Indexical , Indexicals

ConstraintBody ::= Indexicals
| LinExpr RelOp LinExpr
| element(X, CList ,X)
| table(CTable)

Head ::= term  { a compound term with unique variable args }

TellPos ::= Head +: ConstraintBody
TellNeg ::= Head -: ConstraintBody
AskPos ::= Head +? Indexical
AskNeg ::= Head -? Indexical

ConstraintDef ::= TellPos.
? (TellNeg.)
? (AskPos.)
? (AskNeg.)

10.34.11.2 Syntax of Arithmetic Expressions

X ::= variable  { domain variable }
N ::= integer

LinExpr ::= N  { linear expression }
| X
| N * X
| N * N
| LinExpr + LinExpr
| LinExpr - LinExpr

Expr ::= LinExpr
| Expr + Expr
| Expr - Expr
| Expr * Expr
| Expr / Expr  { integer division }
| Expr mod Expr
| Expr rem Expr
| min(Expr ,Expr )
| max(Expr ,Expr )
| abs(Expr )

RelOp ::= #= | #\= | #< | #=< | #> | #>=
10.34.11.3 Operator Declarations

```prolog
:- 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, [/>,/<=]).
```

10.35 Constraint Logic Programming over Rationals or Reals—library([clpq,clpr])

10.35.1 Introduction

The clp(Q,R) system described in this chapter is an instance of the general Constraint Logic Programming scheme introduced by [Jaffar & Michaylov 87]. It is a third-party product, bundled with SICStus Prolog as two library packages. It is not supported by SICS in any way.

The implementation is at least as complete as other existing clp(R) implementations: It solves linear equations over rational or real valued variables, covers the lazy treatment of nonlinear equations, features a decision algorithm for linear inequalities that detects implied equations, removes redundancies, performs projections (quantifier elimination), allows for linear dis-equations, and provides for linear optimization.

10.35.1.1 Referencing this Software

When referring to this implementation of clp(Q,R) in publications, you should use the following reference:


10.35.1.2 Acknowledgments

The development of this software was supported by the Austrian Fonds zur Foerderung der Wissenschaftlichen Forschung under grant P9426-PHY. Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian Federal Ministry for Science and Research.

We include a collection of examples that has been distributed with the Monash University version of clp(R) [Heintze et al. 87], and its inclusion into this distribution was kindly permitted by Roland Yap.