

Part III

Declarative Programming with Constraints

- 1 Introduction to Logic
- 2 Declarative Programming with Prolog
- 3 Declarative Programming with Constraints
- 4 The Semantic Web

Contents

- 3 Declarative Programming with Constraints
 - Motivation
 - CLPFD basics
 - How does CLPFD work
 - FDBG
 - Reified constraints
 - Global constraints
 - Labeling
 - From plain Prolog to constraints
 - Improving efficiency
 - Internal details of CLPFD
 - Disjunctions in CLPFD
 - Modeling
 - User-defined constraints (ADVANCED)
 - Some further global constraints (ADVANCED)
 - Closing remarks

CLPFD – Constraint Logic Programming with Finite Domains

- In this part of the course we get acquainted with CLPFD
 - within the huge area of CP – Constraint Programming
 - we will use Logic Programming, i.e. Prolog
 - for solving Finite Domain Problems
- Examples for other, related approaches:
 - IBM ILog: Constraint Programming on Finite Domains using C++
<https://www.ibm.com/products/ilog-cplex-optimization-studio>
 - SICStus and SWI Prolog have further constraint libraries:
 - CLPR/CLPQ – Constraint Logic Programming on reals/rationals,
 - CLPB – Constraint Logic Programming on Booleans
- CLP(FD) is part of a generic scheme CLP(\mathcal{X}), where \mathcal{X} can also be R, Q, B, etc.
- CLPFD solvers are based on the Constraint Satisfaction Problem (CSP) approach, a branch of Artificial Intelligence (AI)

The structure of CLPFD problems

- Example: a cryptarithmic puzzle such as SEND + MORE = MONEY
- The task: consistently replace letters by different digits so that the equation becomes true (leading zeros are not allowed)
- The (unique) solution: 9567 + 1085 = 10652
- Viewing this task as a CLPFD problem:
 - variables: S, E, N, D, M, O, R, Y
 - variable domains (values allowed): S and M: 1..9, all others 0..9
 - constraints: $S \neq E, S \neq N, \dots, O \neq R, O \neq Y, R \neq Y$, (vars pairwise differ)
 $S*1000+E*100+N*10+D+M*1000+O*100+R*10+E = M*10000+O*1000+N*100+E*10+Y$
- A CLPFD task, as a mathematical problem, consists of:
 - variables X_1, \dots, X_n
 - domains D_1, \dots, D_n , each being a finite set of integers (variable X_i can only take values from its domain, D_i , i.e. $X_i \in D_i$)
 - constraints (relations) between X_i -s that have to be satisfied, e.g. $X_1 \neq X_2, X_2 + X_3 = X_5$, etc.
- Solving a task requires assigning each variable a value from its domain so that all the constraints are satisfied (to obtain one/all solutions, possibly maximizing some variables, etc.)

SEND MORE MONEY – Prolog and CLPFD solutions

Prolog: **generate** and **test** (check)

```
:- use_module(library(between)).
send0(SEND, MORE, MONEY) :-
    Ds = [S,E,N,D,M,O,R,Y],
    maplist(between(0, 9), Ds),
    alldiff(Ds),
    S =\= 0, M =\= 0,
    SEND is 1000*S+100*E+10*N+D,
    MORE is 1000*M+100*O+10*R+E,
    MONEY is
        10000*M+1000*O+100*N+10*E+Y,
    SEND+MORE == MONEY.

% alldiff(+L):
% elements of L are all different
alldiff([]).
alldiff([D|Ds]) :-
    \+ member(D, Ds), alldiff(Ds).
```

Run time: 13.1 sec

CLPFD: **test** (**constrain**) and **generate**

```
:- use_module(library(clpfd)).
send_clpfd(SEND, MORE, MONEY) :-
    Ds = [S,E,N,D,M,O,R,Y],
    domain(Ds, 0, 9),
    all_different(Ds),
    S #\= 0, M #\= 0,
    SEND #= 1000*S+100*E+10*N+D,
    MORE #= 1000*M+100*O+10*R+E,
    MONEY #=
        10000*M+1000*O+100*N+10*E+Y,
    SEND+MORE #= MONEY,
    labeling([], Ds).
```

New implementation features needed:

- associating a **domain** with a variable
- **constraints** performing repetitive pruning

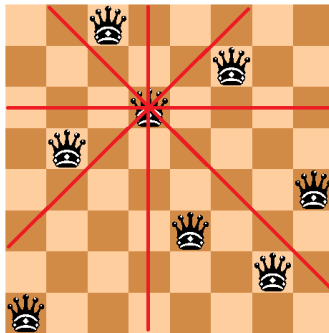
Run time: 0.00011 sec

The CLPFD approach

- Calling a constraint is called **posting**
- A **constraint** can be of two kinds:
 - **primitive**: prunes the domain (set of poss. values) of a var. and exits: e.g. $S \# \neq 0$ simply removes 0 from the domain of s and exits
 - **composite**: performs an initial pruning, and then becomes a **daemon**, e.g. $SEND \# = 1000*S+100*E+10*N+D$
 - 1 waits in the background (sleeps) until there is a change in the domain of one of its variables
 - 2 wakes up to possibly prune the domain of other variables (*in forward Prolog execution domains never grow, hence we speak of pruning or narrowing of domains*)
 - 3 if the constraint is now bound to fail, it initiates a backtrack
 - 4 if the constraint is now bound to hold, it exits with success
 - 5 otherwise goes to step 1.
- When all constraints are **posted**, the search phase, **labeling**, is started:
 - **labeling** repeatedly selects a var. and creates a choice point for it
 - prunes the domain of the var., causing constraints to wake up
 - eventually makes all variables bound, and thus finds solutions

Another CLPFD example: the N-queens problem

- Place N queens on an $N \times N$ chessboard, so that no two queens attack each other



- The Prolog list $[Q_1, \dots, Q_N]$ is a compact representation of a placement: row i contains a queen in column q_i , for each $i = 1, \dots, N$.
- The list encoding the above placement: $[3, 6, 4, 2, 8, 5, 7, 1]$
- Note that this **modeling** of the problem in itself ensures that no two queens are present in any given row

Constraints in the N-queens problem

- It is enough to ensure that no queen threatens other queens **below** it (as the “threatens” relation is symmetrical)
- Queen q threatens positions marked with *

	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Q_1	Q					Q					Q				
Q_2	*	*				*	*	*				*	*	*	
Q_3	*		*			*		*			*	*	*	*	
Q_4	*			*		*		*			*		*	*	
Q_5	*				*	*		*			*		*	*	

- Assume $j < k$, and let $I = k - j$. Queen q_j threatens q_k iff

$$Q_k = Q_j + I, \quad \text{or} \quad Q_k = Q_j - I, \quad \text{or} \quad Q_k = Q_j$$
- The Prolog code for checking that two queens **do not** threaten each other:

```
% no_threat(QJ, QK, I): queens placed in column QJ of row m and
%                               in column QK of row m+I
% do not threaten each other.
no_threat(QJ, QK, I) :-
    QK =\= QJ+I, QK =\= QJ-I, QK =\= QJ.
```

Constraints in the N-queens problem (contd.)

- Doubly nested loop needed: check each queen w.r.t. each queen below it
- The structure of the code, demonstrated for the 4 queens case:

```
queens4([Q1,Q2,Q3,Q4]) :-
  % Queen Q1 does not threaten the queens Q2, Q3, Q4 below it:
  no_threat(Q1, Q2, 1), no_threat(Q1, Q3, 2), no_threat(Q1, Q4, 3),
  % Queen Q2 does not threaten the queens Q3, Q4 below it:
  no_threat(Q2, Q3, 1), no_threat(Q2, Q4, 2),
  no_threat(Q3, Q4, 1). % Queen Q3 does not threaten queen Q4 below it
```

- An inner loop can be implemented via this predicate:


```
% no_attack(Q, Qs, I): Q is the placement of the queen in row m,
% Qs lists the placements of queens in rows m+I, m+I+1, ...
% Queen in row m does not attack any of the queens listed in Qs.
no_attack(_, [], _).
no_attack(X, [Y|Ys], I):-
  no_threat(X, Y, I), J is I+1, no_attack(X, Ys, J).
```
- Using no_attack/3, the 4 queens case can be simplified to:

```
queens4([Q1,Q2,Q3,Q4]) :-
  no_attack(Q1, [Q2,Q3,Q4], 1),
  no_attack(Q2, [Q3,Q4], 1),
  no_attack(Q3, [Q4], 1).
```

Plain Prolog solution: “generate and test”

```
% queens_gt(N, Qs): Qs is a valid placement of N queens on an NxN chessboard.
queens_gt(N, Qs):-
  length(Qs, N), maplist(between(1, N), Qs), safe(Qs).

% safe(Qs): In placement Q, no pair of queens attack each other.
safe([]).
safe([Q|Qs]):-
  no_attack(Q, Qs, 1), safe(Qs).

% no_attack(Q, Qs, I): Q is the placement of the queen in row k,
% Qs lists the placements of queens in rows k+I, k+I+1, ...
% Queen in row k does not attack any of the queens listed in Qs.
no_attack(_, [], _).
no_attack(X, [Y|Ys], I):-
  no_threat(X, Y, I), J is I+1, no_attack(X, Ys, J).

% no_threat(X, Y, I): queens placed in column X of row k and in
% column Y of row k+I
% do not attack each other.
no_threat(X, Y, I) :-
  Y =\= X, Y =\= X-I, Y =\= X+I.
```

Evaluation

- Nice solution: declarative, concise, easy to validate
- But...

N	Time for all solutions in msec (on an Intel i3-3110M, 2.40GHz CPU)
4	0
5	16
6	46
7	515
8	10,842
9	275,170
10	7,926,879
15	~ 10,000 years
20	~ 1000 bn years

Contents

- 3 Declarative Programming with Constraints
 - Motivation
 - CLPFD basics
 - How does CLPFD work
 - FDBG
 - Reified constraints
 - Global constraints
 - Labeling
 - From plain Prolog to constraints
 - Improving efficiency
 - Internal details of CLPFD
 - Disjunctions in CLPFD
 - Modeling
 - User-defined constraints (ADVANCED)
 - Some further global constraints (ADVANCED)
 - Closing remarks

The main steps of solving a CSP/CLPFD problem

- **Modeling** – transforming the problem to a CSP
 - defining the variables and their domains
 - identifying the constraints between the variables
- **Implementation** – the structure of the CSP program
 - Set up variable domains: `N in {1,2,3}, domain([X,Y], 1, 5)`.
 - Post constraints. Preferably, no choice points should be created.
 - Label the variables, i.e. systematically explore all variable settings.
- **Optimization** – **redundant** constraints, labeling heuristics, constructive disjunction, shaving, etc.

library(clpfd) – basic concepts

- To load the library, place this directive at the beginning of your program:


```
:- use_module(library(clpfd)).
```
- **Domain**: a finite set of integers (allowing the restricted use of infinite intervals for convenience)
- **Constraints**:
 - membership, e.g. `X in 1..5` ($1 \leq X \leq 5$)
 - arithmetic, e.g. `X #< Y+1` ($X < Y + 1$)
 - reified, e.g. `X#<Y+5 #<=> B` (B is the truth value of $X < Y + 5$)
 - propositional, e.g. `B1 #\ B2` (at least one of the two Boolean values B1 and B2 is true)
 - combinatorial, e.g. `all_distinct([V1,V2,...])` (variables [V1,V2,...] are pairwise different)
 - user-defined
- Two main variants: **formula** constraints and **global** constraints
- **Formula** constraints are written using operators, while **global** constraints use the canonical Prolog term format.
- Global constraints operate on lists of variables, most of the time.

Membership constraints

- `domain(+Vars, +Min, +Max)` where
 - Min: `<integer>` or `inf(-∞)`,
 - Max: `<integer>` or `sup(+∞)`:
 All elements of list `Vars` belong to the interval `[Min,Max]`.
 Example: `domain([A,B,C], 1, sup)` – variables A, B and C are positive

- `X in +ConstRange`: X belongs to the set `ConstRange`, where:

```

ConstantSet ::= {<integer>, ..., <integer>}
Constant    ::= <integer> | inf | sup
ConstRange ::= ConstRange
              | Constant .. Constant           (interval)
              | ConstRange /\ ConstRange      (intersection)
              | ConstRange \/ ConstRange      (union)
              | \ ConstRange                   (complement)
    
```

Examples:

```

A in inf .. -1,      B in \ (0 .. sup),      C in {1,4,7,2}.
    
```

Arithmetic formula constraints

- In the division and remainder operations below *truncated* means rounded towards 0, while *floored* means rounded towards $-\infty$
- Arithmetic formula constraints: `Expr RelOp Expr` where

```
RelOp ::= #= | #\= | #< | #=< | #> | #>=
```

```

Expr ::= <integer> | <variable>
        | - Expr | Expr + Expr | Expr - Expr | Expr * Expr
        | Expr / Expr           % truncated integer division
        | Expr // Expr          % // ≡ /
        | Expr div Expr         % floored integer division
        | Expr rem Expr         % truncated remainder
        | Expr mod Expr         % floored remainder
        | min(Expr, Expr)
        | max(Expr, Expr)
        | abs(Expr)
    
```

Global arithmetic constraints

- `sum(+Xs, +RelOp, ?Value)`: $\sum Xs \text{ RelOp Value}$.
- `scalar_product(+Coeffs, +Xs, +RelOp, ?Value [, +Options])`
(last arg. optional): $\sum_i \text{Coeffs}_i * Xs_i \text{ RelOp Value}$.
where `Coeffs` has to be a list of **integers**. Examples:

$$\text{scalar_product}([1,2,5], [X,Y,Z], \#<, U) \equiv X + 2*Y + 5*Z \#< U$$

$$\text{scalar_product}([1,1,1], [X,Y,Z], \#=:, U) \equiv \text{sum}([X,Y,Z], \#=:, U)$$
- `minimum(?V, +Xs), maximum(?V, +Xs)`: `V` is the minimum/maximum of the elements of the list `Xs`. Example:

$$\text{minimum}(M, [X,Y,Z]) \equiv \text{min}(X, \text{min}(Y,Z)) \#=: M$$

Relational symbols

- Standard Prolog relations and CLPFD relations should not be confused; their meaning is in general quite different
- Example: “equals”
 - `Expr1#=Expr2`: post a constraint that `Expr1` and `Expr2` must be equal
 - `Term1=Term2`: attempt to unify `Term1` and `Term2`
 - `domain([A,B],3,4), A+1#=B. \implies A=3, B=4`
 - `domain([A,B],3,4), A+1=B. \implies Type error`
(This tries to unify `B` with the compound `A+1`. As domain variables can only be unified with integers, an error is raised)
- Example: “less than”
 - `Expr1#<Expr2`: post a constraint that `Expr1` must be less than `Expr2`
 - `Expr1<Expr2`: checks if `Expr1` is less than `Expr2`
 - `domain([A,B],3,4), A#<B. \implies A=3, B=4`
 - `domain([A,B],3,4), A<B. \implies Instantiation error`
(arguments in arithmetic comparison BIPs must be ground)

Global constraints

- Some global constraints:
 - `all_different([X1, ..., Xn])`: same as `Xi #\= Xj` for all $1 \leq i < j \leq n$.
 - `all_distinct([X1, ..., Xn])`: same as `all_different`, but does much better pruning (guarantees so called **domain-consistency**, see later)
- ```
| ?- L=[A,B,C], domain(L, 1, 2), all_different(L).
 \implies A in 1..2, B in 1..2, C in 1..2
| ?- L=[A,B,C], domain(L, 1, 2), all_distinct(L).
 \implies no
```
- And many many more...

## Labeling – at a glance

- In general, there are multiple solutions  $\implies$  labeling is necessary (Even if there is a single solution, it often cannot be inferred directly from the constraints)
  - Labeling: search by creating choice points and systematic assignment of feasible values to variables
  - During labeling, narrowing the domain of a variable may wake up constraints that in turn may prune the domain of other variables etc. This is called **propagation**.
  - `indomain(?Var)`: for variable `Var`, its feasible values are assigned one after the other (in ascending order)
  - `labeling(+Options, +Vars)`: assigns values to all variables in `Vars`. The options control, for example, the order in which
    - variables are selected for labeling
    - the feasible values of the selected variable are tried
- Most of the options impact only the efficiency of the algorithm, not its correctness.

## N-queens – the Prolog solution (recall)

```

% Qs is a valid placement of N queens on an NxN chessboard.
queens_gt(N, Qs):-
 length(Qs, N), maplist(between(1, N), Qs), safe(Qs), true.

% safe(Qs): In placement Q, no pair of queens attack each other.
safe([]).
safe([Q|Qs]):-
 no_attack(Q, Qs, 1), safe(Qs).

% no_attack(Q, Qs, I): Q is the placement of the queen in row k,
% Qs lists the placements of queens in rows k+I, k+I+1, ...
% Queen in row k does not attack any of the queens listed in Qs.
no_attack(_, [], _).
no_attack(X, [Y|Ys], I):-
 no_threat(X, Y, I), J is I+1, no_attack(X, Ys, J).

% no_threat(X, Y, I): queens placed in column X of row k and in column Y of row k+I
% do not attack each other.
no_threat(X, Y, I) :-
 Y #\= X, Y #\= X-I, Y #\= X+I.

```

## N-queens – the CLPFD solution

```

% Qs is a valid placement of N queens on an NxN chessboard.
queens_fd(N, Qs):-
 length(Qs, N), domain(Qs, 1, N), safe(Qs), labeling([ff],Qs).

% safe(Qs): In placement Q, no pair of queens attack each other.
safe([]).
safe([Q|Qs]):-
 no_attack(Q, Qs, 1), safe(Qs).

% no_attack(Q, Qs, I): Q is the placement of the queen in row k,
% Qs lists the placements of queens in rows k+I, k+I+1, ...
% Queen in row k does not attack any of the queens listed in Qs.
no_attack(_, [], _).
no_attack(X, [Y|Ys], I):-
 no_threat(X, Y, I), J is I+1, no_attack(X, Ys, J).

% no_threat(X, Y, I): queens placed in column X of row k and in column Y of row k+I
% do not attack each other.
no_threat(X, Y, I) :-
 Y #\= X, Y #\= X-I, Y #\= X+I.

```

## Evaluation

Time for all solutions in msec (on an Intel i3-3110M, 2.40GHz CPU):

| N  | Prolog         | CLPFD   |
|----|----------------|---------|
| 4  | 0              | 0       |
| 5  | 16             | 0       |
| 6  | 46             | 0       |
| 7  | 515            | 0       |
| 8  | 10,842         | 0       |
| 9  | 275,170        | 31      |
| 10 | 7,926,879      | 94      |
| 11 | ~ 2 days       | 421     |
| 12 | ~ 2 months     | 2,168   |
| 13 | ~ 6 years      | 10,982  |
| 14 | ~ 250 years    | 54,242  |
| 15 | ~ 10,000 years | 351,424 |

## A simple practice task

Write a predicate that enumerates the solutions of the following task

```

% incr(L, Len, N): L is a strictly increasing list of length Len,
% containing integers in 1..N.
| ?- incr(L, 3, 3). ---> L = [1,2,3] ; no
| ?- incr(L, 3, 4). ---> L = [1,2,3] ; L = [1,2,4] ;
 L = [1,3,4] ; L = [2,3,4] ; no
| ?- incr(L, 2, 5), L = [3|_]. ---> L = [3,4] ; L = [3,5] ; no

```

A solution:

```

incr(L, Len, N) :-
 length(L, Len), % Determining the variables
 domain(L, 1, N), % Setting up the domains
 L = [H|T], incr_list(T, H), % Posting the constraints
 labeling([], L). % Labeling

incr_list([X2|T], X1) :-
 X1 #< X2, incr_list(T, X2).

incr_list([], _).

```

## Contents

- 3 Declarative Programming with Constraints
  - Motivation
  - CLPFD basics
  - How does CLPFD work
  - FDBG
  - Reified constraints
  - Global constraints
  - Labeling
  - From plain Prolog to constraints
  - Improving efficiency
  - Internal details of CLPFD
  - Disjunctions in CLPFD
  - Modeling
  - User-defined constraints (ADVANCED)
  - Some further global constraints (ADVANCED)
  - Closing remarks

## Infeasible values

- A constraint  $C$  is implemented by a **daemon**, which ensures that  $C$  holds
- Consider the constraint  $x+5 \neq Y$ , which represents the relation  $r = \{(x, y) \mid x + 5 = y\} = \{\dots, \langle -1, 4 \rangle, \langle 0, 5 \rangle, \langle 1, 6 \rangle, \langle 2, 7 \rangle, \dots\}$
- The CLPFD constraint  $x+5 \neq Y$  has to ensure that  $r(x, Y)$  holds:
  - 1 if both  $x$  and  $Y$  are bound : check if  $\langle x, Y \rangle \in r$  holds, i.e.  $x+5=Y$
  - 2 if only  $x$  is bound: set  $Y$  to  $x+5$ , if possible, else fail
  - 3 if only  $Y$  is bound: set  $x$  to  $Y-5$  if possible, else fail
  - 4 if  $x$  and  $Y$  are unbound: remove **infeasible** values from their domains:  
E.g.:  $X$  in  $1..6$ ,  $Y$  in  $\{1, 6, 7, 9\}$ , Infeasible for  $x: 3, 5, 6$ ; for  $Y: 1$   
(case 4 covers 1–3 as well, assuming empty domain  $\Rightarrow$  failure)
- Let  $D(u)$  denote the domain of variable  $u$ .  
With respect to a constraint/relation  $r(x, y)$ :  
 $a \in D(x)$  is **infeasible** iff there is no  $b \in D(y)$  such that  $r(a, b)$  holds;  
 $b \in D(y)$  is **infeasible** iff there is no  $a \in D(x)$  such that  $r(a, b)$  holds
- **In general:** A value  $d_i \in D(x_i)$  is **infeasible** w.r.t.  $r(x_1, \dots, x_i, \dots)$ , if no assignment can be found for the remaining variables – mapping each  $x_j, j \neq i$  to some  $d_j \in D(x_j)$  – so that  $r(d_1, \dots, d_i, \dots)$  holds

## Implementation of constraints

- The main data structure: the **backtrackable constraint store** – maps variables to their domains.
- **Simple** constraints: e.g.  $X$  in  $\text{inf}..9$  or  $X \#< 10$  modify the store and exit, e.g. add  $X \#< 10$  to store  $X$  in  $5..20 \implies X$  in  $5..9 (= \text{inf}..9 \cap 5..20)$
- **Composite** constraints are implemented as **daemons**, which keep removing **infeasible** values from argument domains
- Example store content:  $X$  in  $1..6$ ,  $Y$  in  $\{1, 6, 7, 9\}$ 
  - Daemon for  $x+5 \neq Y$  may remove 3, 5, 6 from  $x$  and 1 from  $Y$
  - Resulting store content:  $X$  in  $\{1, 2, 4\}$ ,  $Y$  in  $\{6, 7, 9\}$
- A constraint  $C$  is said to be **entailed** (or implied) by the store iff:
  - $C$  holds for **ANY** variable assignment allowed by the store
- For example, store  $X$  in  $\{1, 2\}$ ,  $Y$  in  $\{6, 7\}$  does **not** entail  $x + 5 \neq Y$ , as the constraint does not hold for the assignment  $x = 1$ ,  $Y = 7$
- However, store  $X$  in  $\{1\}$ ,  $Y$  in  $\{6\}$  does entail  $x + 5 \neq Y$ , and store  $U$  in  $5..10$ ,  $V$  in  $30..40$  entails  $2*U+9 \#< V$
- A daemon may **exit** (die), when its constraint is **entailed** by the store (as entailment implies that the constraint will never be able to do any pruning)

## Strength of reasoning for composite constraints

- **Domain-consistency**, also called **arc-consistency**:  
all infeasible values are removed
  - Example store:  $X$  in  $0..6$ ,  $Y$  in  $\{1, 6, 8, 9\}$ 
    - Daemon for  $X+5 \neq Y$  removes 0, 2, 5, 6 from  $x$  and 1 from  $Y$
    - Resulting store:  $X$  in  $\{1, 3, 4\}$ ,  $Y$  in  $\{6, 8, 9\}$
  - Cost: **exponential** in the number of variables
- **Bound-consistency**: reasoning views domains as intervals, only removes bounds, possibly repeatedly  
(a *middle* element, such as 2 in the domain of  $x$  above, is not removed)
  - Weaker than domain-consistency, examples:
    - store:  $X$  in  $0..6$ ,  $Y$  in  $\{1, 6, 8, 9\}$ , constraint  $X+5 \neq Y \implies$  removes 0, 6 and 5 from  $x$ , and 1 from  $Y$  (**2 is kept in  $x$** )  
new store:  $X$  in  $1..4$ ,  $Y$  in  $\{6, 8, 9\}$
    - $X$  in  $1..6$ ,  $Y$  in  $\{100, 200\}$ ,  $Z$  in  $\text{inf}..sup$ , constraint  $X+Y \neq Z \implies$   
only  $z$  is pruned:  $Z$  in  $101..206$  (**107..200 are not feasible**)
  - Cost: **linear** in the number of variables

## Bound-consistency, further details (ADVANCED)

- Bound-consistency relies on the interval closure of the store, obtained by removing all ‘holes’ from the domains:
  - Store:  $S_0 = A \text{ in } \{0,1,2,3,4,6\}, V \text{ in } \{-1,1,3,4,5\}$
  - Interval closure of the store:  $\mathcal{IC}(S_0) = A \text{ in } 0..6, V \text{ in } -1..5$
- In general: the interval closure of the store maps each variable  $x$  to  $\text{MinX}.. \text{MaxX}$ , where  $\text{MinX}/\text{MaxX}$  is the smallest/largest value in  $x$ 's domain
- Bound-consistency reasoning repeatedly removes all boundary values that are infeasible w.r.t. the interval closure of the store
- Example:  $A \# = \text{abs}(V)$  in store  $S_0$ :
 

```
| ?- A in (0..4)\{6}, V in {-1}\{1}\(3..5), A # = abs(V).
=> A in 0..4, V in {-1}\{1}\(3..4) ?
```

  - boundary value 6 is removed from the domain of A, as  $v$  cannot be 6 nor  $-6$  in  $\mathcal{IC}(S_0) \Rightarrow S_1 = A \text{ in } 0..4, V \text{ in } \{-1,1,3,4,5\}$
  - boundary value 5 is removed from  $v$ , as A cannot be 5 in  $\mathcal{IC}(S_1) \Rightarrow S_2 = A \text{ in } 0..4, V \text{ in } \{-1,1,3,4\}$
  - A's boundary value 0 is kept, as in  $\mathcal{IC}(S_2)$   $v$ 's domain is  $-1..4 \ni 0$

## Consistency levels guaranteed by SICStus Prolog

- Membership constraints (trivially) ensure domain-consistency.
- Linear arithmetic constraints ensure at least bound-consistency.
- Nonlinear arithmetic constraints do not guarantee bound-consistency.
- For all constraints, when all the variables of the constraint are bound, the constraint is guaranteed to deliver the correct result (success or failure).
 

```
| ?- X in {4,9}, Y in {2,3}, Z # = X-Y. => Z in 1..7 ?
 Bound consistent
```

```
| ?- X in {4,9}, Y in {2,3},
 scalar_product([1,-1], [X,Y], # =, Z, [consistency(domain)]).
/* not available in SWI, scalar_product can only have 4 arguments*/
=> Z in(1..2)\(6..7) ?
 Domain consistent
```

```
| ?- domain([X,Y],-9,9), X*X+2*X+1 # = Y. => X in -4..4, Y in -7..9 ?
 Not even bound consistent
```

```
| ?- domain([X,Y],-9,9), (X+1)*(X+1)# = Y. => X in -4..2, Y in 0..9 ?
 Bound consistent
```

## Implementation of constraints

- A constraint  $C$  is implemented by:
  - transforming  $C$  (possibly at compile time) to a series of elementary constraints,
    - e.g.  $X*X \# > Y \Rightarrow A \# = X*X, A \# > Y$  (formula constraints only).
  - posting  $C$ , or each of the primitive constraints obtained from  $C$
- To see the the pending constraints in SICStus execute the code below (pending constraints are always shown in SWI):
 

```
| ?- assert(clpfd:full_answer).
```
- Examples (with some editing for better readability):

### SICStus Prolog

```
| ?- domain([X,Y],-9,9), X*X+2*X+1# = Y.
A# = X*X,
Y# = 2*X+A+1,
X in -4..4,
Y in -7..9,
A in 0..16 ?
```

### SWI Prolog

```
?- [X,Y] ins -9..9, X*X+2*X+1# = Y.
2*X# = B, X^2# = A, B+A# = C, C+1# = Y,
X in -4..4, A in 0..16,
B in -8..8, C in -8..8,
Y in -7..9.
```

## Execution of constraints

To execute a constraint  $C$ :

- execute completely (e.g.  $X \# < 3$ ); or
- create a daemon for  $C$ :
  - specify the **activation conditions** (how to set the ‘‘alarm clock’’ to wake up the daemon)
  - prune the domains*
  - until** the **termination condition** becomes true **do** go to sleep (wait for activation)
  - prune the domains*
  - enduntil**

- A  $\# \setminus =$  B (domain-consistent)
  - Activation:** when A or B is instantiated.
  - Pruning:** remove the value of the instantiated variable from the domain of the other.
  - Termination:** when A or B is instantiated.
  - Example:** `| ?- A in 1..5, A # \ = B, B = 3.`

## Execution of constraints

To execute a constraint  $C$ :

- execute completely (e.g.  $X \# < 3$ ); or
- create a daemon for  $C$ :
  - specify the **activation conditions** (how to set the ‘‘alarm clock’’ to wake up the daemon)
  - prune the domains*
  - until** the **termination condition** becomes true **do** go to sleep (wait for activation)
  - prune the domains*
  - enduntil**

- A  $\# \setminus =$  B (domain-consistent)
  - Activation:** when A or B is instantiated.
  - Pruning:** remove the value of the instantiated variable from the domain of the other.
  - Termination:** when A or B is instantiated.
  - Example:** `| ?- A in 1..5, A # \ = B, B = 3.`



## Execution of constraints, continued

- **Activation condition:** the domain of a variable  $x$  changes in SOME way  
SOME can be:
  - Any change of the domain
  - Lower bound change
  - Upper bound change
  - Lower or upper bound change
  - Instantiation
  - ...
- The **termination condition** is constraint specific
  - **earliest:** when the constraint is **entailed** by the constraint store  
i.e. it is bound to hold in the given constraint store
  - **latest:** when all its variables are instantiated
  - In most of the cases it does **not** pay off waking up a constraint quite often, just to check if it can terminate...

## Implementation of some constraints

- $A \#< B$  (domain-consistent)
  - **Activation:** when  $\min(A)$  (the lower bound of  $A$ ) or  
when  $\max(B)$  (the upper bound of  $B$ ) changes.
  - **Pruning:**  
(the highest feasible value for  $A$ , given  $B$ 's domain?  $\max(B)-1$ )  
(the lowest feasible value for  $B$ , given  $A$ 's domain?  $\min(A)+1$ )  
remove from the domain of  $A$  all integers  $\geq \max(B)$  ( $\max(B)..sup$ )  
remove from the domain of  $B$  all integers  $\leq \min(A)$  ( $inf..min(A)$ )
  - **Termination:** when one of  $A$  and  $B$  is instantiated (not optimal)
  - **Example:** `| ?- domain([A,B], 1, 5), A #< B, B in 1..4, A = 2.`

## Implementation of some constraints (contd.)

- $X+Y \#= T$  (bound-consistent)
  - **Activation:** at lower or upper bound change of  $X$ ,  $Y$ , or  $T$ .
  - **Pruning:**  
(the lowest possible  $T$ , given the domains of  $X$  and  $Y$ ?  $\min(X)+\min(Y)$ )  
narrow the domain of  $T$  to  $(\min(X)+\min(Y))..(\max(X)+\max(Y))$   
(the lowest possible  $X$ , given the domains of  $T$  and  $Y$ ?  $\min(T)-\max(Y)$ )  
narrow the domain of  $X$  to  $(\min(T)-\max(Y))..(\max(T)-\min(Y))$   
narrow the domain of  $Y$  to  $(\min(T)-\max(X))..(\max(T)-\min(X))$
  - **Termination:** if all three variables are instantiated (after the pruning)
  - **Example:** `| ?- domain([X,Y,T], 1, 5), T #= X+Y, X #> 2.`
- `all_distinct([A1,...])` (domain-consistent)
  - **Activation:** at any domain change of any variable.
  - **Pruning:** remove all infeasible values from the domains of all variables (using an algorithm based on maximal matchings in bipartite graphs)
  - **Termination:** when at most one of the variables is uninstantiated.
  - **Example:** `| ?- L=[W,X,Y,Z], domain(L,1,4), all_distinct(L), W#<3, Z#<3.`

## Interplay of multiple constraints

- A simple example:  
`| ?- domain([X,Y], 0, 100), X+Y #= 10, X-Y #= 4.`  
 $\implies X \text{ in } 4..10, Y \text{ in } 0..6$
- Another example:  
`| ?- domain([X,Y], 0, 100), X+Y #= 10, X+2*Y #= 14.`  
 $\implies X = 6, Y = 4$
- More examples in the practice tool <https://ait.plwin.dev/C1-1>

### 3 Declarative Programming with Constraints

- Motivation
- CLPFD basics
- How does CLPFD work
- **FDBG**
- Reified constraints
- Global constraints
- Labeling
- From plain Prolog to constraints
- Improving efficiency
- Internal details of CLPFD
- Disjunctions in CLPFD
- Modeling
- User-defined constraints (ADVANCED)
- Some further global constraints (ADVANCED)
- Closing remarks

- Created as an MSc Thesis by Dávid Hanák and Tamás Szeredi at Budapest University of Technology and Economics back in 2001
- Now part of SICStus
- Shows details of all important CLPFD events
  - Constraints waking up
  - Pruning
  - Constraints exiting
  - Labeling
- Highly customizable
- Output can be written to a file

### Example: tracking the life-cycle of constraints

```
| ?- use_module([library(clpfd),library(fdbg)]).
| ?- Xs=[X1,X2], fdbg_assign_name(Xs, 'X'), fdbg_on, domain(Xs, 1, 6),
 X1+X2 #= 8, X2 #>= 2*X1+1.

domain(<X_1>,<X_2>],1,6) X_1 = inf..sup -> 1..6
 X_2 = inf..sup -> 1..6
 Constraint exited.

<X_1>+<X_2> #= 8 X_1 = 1..6 -> 2..6
 X_2 = 1..6 -> 2..6

<X_2> #>= 2*<X_1>+1 X_1 = 2..6 -> {2}
 X_2 = 2..6 -> 5..6
 Constraint exited.

<X_1>+<X_2> #= 8 X_1 = {2}
 X_2 = 5..6 -> {6}
 Constraint exited.

Xs = [2,6], X1 = 2, X2 = 6 ?
```

(This example is available as <https://ait.plwin.dev/C1-1/c.>)

### Example: labeling

```
| ?- X in 1..3, labeling([bisect], [X]).
<fdvar_1> in 1..3
 fdvar_1 = inf..sup -> 1..3
 Constraint exited.

Labeling [2, <fdvar_1>]: starting in range 1..3.
Labeling [2, <fdvar_1>]: bisect: <fdvar_1> <= 2

Labeling [4, <fdvar_1>]: starting in range 1..2.
Labeling [4, <fdvar_1>]: bisect: <fdvar_1> <= 1

X = 1 ? ;
Labeling [4, <fdvar_1>]: bisect: <fdvar_1> >= 2

X = 2 ? ;
Labeling [4, <fdvar_1>]: failed.

Labeling [2, <fdvar_1>]: bisect: <fdvar_1> >= 3

X = 3 ? ;
Labeling [2, <fdvar_1>]: failed.

no
```

# Contents

- 3 Declarative Programming with Constraints
  - Motivation
  - CLPFD basics
  - How does CLPFD work
  - FDBG
  - Reified constraints
  - Global constraints
  - Labeling
  - From plain Prolog to constraints
  - Improving efficiency
  - Internal details of CLPFD
  - Disjunctions in CLPFD
  - Modeling
  - User-defined constraints (ADVANCED)
  - Some further global constraints (ADVANCED)
  - Closing remarks

# Reification – introductory example

- Given  $X \text{ in } 0..9, Y \text{ in } 0..9$ , define constraint “exactly one of  $x$  and  $y$  is  $> 0$ ”
- Hint: let the 0-1 variable  $x_p$  (for  $x$  Positive) reflect the truth value of  $x \#> 0$ .
- Use the // integer division op to define this relationship between  $x$  and  $x_p$

$$x_p \# = (x+9) // 10$$

- With this trick it is easy to achieve our goal:

```

exactly_1_pos(X, Y) :- X in 0..9, Y in 0..9,
 (X+9)//10 \#= Xp, (Y+9)//10 \#= Yp, Xp+Yp \#= 1.
| ?- X \#> 3, exactly_1_pos(X, Y). => Y = 0
| ?- Y \# = 0, exactly_1_pos(X, Y). => X in 1..9

```

- Constraint  $x_p \# = (x+9) // 10$  reflects (or **reifies**) the truth value of  $x \#> 0$  in the boolean variable  $x_p$
- `library(clpfd)` supports **reified constraints** using this syntax:

```

X \#> 0 \#<=> Xp or in general:
<reifiable constraint> \#<=> B

```

This does not rely on knowing the domain of  $x$ .  
 (SWI Prolog CLPFD uses the  **$\#<==>$**  operator instead of  **$\#<=>$** )

# Reification – what is it?

- Reification = reflecting the truth value of a constraint into a 0/1-variable
- Form:  $c \#<=> B$  (in SWI  $\#<==>$ ), where  $c$  is a *reifiable* constraint and  $B$  is a 0/1-variable
- Meaning:  $c$  holds if and only if  $B=1$
- E.g.:  $(X \#> 5) \#<=> B$  ( $X > 5$  holds iff  $B$  is true ( $B = 1$ )) (\*)
- Four implications:
  - If  $c$  holds, then  $B$  must be 1
  - If  $B=1$ , then  $c$  must hold
  - If  $\neg c$  holds, then  $B$  must be 0
  - If  $B=0$ , then  $\neg c$  must hold
- Which constraints can be reified?
  - Arithmetic formula constraints ( $\# =, \# < =,$  etc.) **can** be reified
  - The  $X \text{ in } ConstRange$  membership constraint **can** be reified, e.g. rewrite (\*) to a membership constraint:  $(X \text{ in } 6..sup) \#<=> B$
  - In SICStus, `scalar_product` **can** be reified
  - All other global constraints (e.g. `all_different/1, sum/3`) **cannot** be reified: `all_different([X,Y]) \#<=> B` causes an error
- Having introduced Boolean vars, it's feasible to allow propositional ops

# Propositional constraints – working with Boolean variables

- Propositional connectives allowed by SICStus Prolog CLPFD:

| Format                | Meaning      | Priority | Kind | SWI notation    |
|-----------------------|--------------|----------|------|-----------------|
| $\# \backslash Q$     | negation     | 710      | fy   | (same)          |
| $P \# \backslash Q$   | conjunction  | 720      | yfx  | (same)          |
| $P \# \backslash Q$   | exclusive or | 730      | yfx  | (same)          |
| $P \# \backslash / Q$ | disjunction  | 740      | yfx  | (same)          |
| $P \# => Q$           | implication  | 750      | xfy  | $P \# ==> Q$    |
| $Q \# < = P$          | implication  | 750      | yfx  | $Q \# < == P$   |
| $P \# < = > Q$        | equivalence  | 760      | yfx  | $P \# < == > Q$ |

- The operand of a propositional constraint can be
  - a variable  $B$ , whose domain automatically becomes  $0..1$ ; or
  - an integer (0 or 1); or
  - a reifiable constraint; or
  - recursively, a propositional constraint
- Example:  $(X \#> 5) \# \backslash (Y \#> 7)$   
 implemented via reification:  $(X \#> 5) \#<=> B1, (Y \#> 7) \#<=> B2, B1 \# \backslash B2$
- Note that reification is a special case of **equivalence**

## Using 0/1-variables in arithmetic constraints

- 0/1-variables can be used just like any other FD-variable, e.g., in arithmetic calculations
- Typical usage: counting the number of times a given constraint holds
- Example:

```
% pcount(L, N): list L has N positive elements.
pcount([], 0).
pcount([X|Xs], N) :-
 (X #> 0) #<=> B,
 N #= N1+B,
 pcount(Xs, N1).
```

## Executing reified constraints

- Recall: a constraint  $C$  is said to be **entailed** (or implied) by the store:
  - iff  $C$  holds for any variable assignment allowed by the store
  - e.g.: store  $X$  in  $5..10$ ,  $Y$  in  $12..15$  entails the constraint  $X \#< Y$  as for arbitrary  $X$  in  $5..10$  and arbitrary  $Y$  in  $12..15$ ,  $X \#< Y$  holds
- Posting the constraint  $C \#<=> B$  immediately enforces  $B$  in  $0..1$
- The execution of  $C \#<=> B$  requires three daemons:
  - When  $B$  is **instantiated**:
    - if  $B=1$ , **post**  $C$ ; if  $B=0$ , **post**  $\neg C$
  - When  $C$  is **entailed**, **set**  $B$  to 1
  - When  $C$  is **disentailed** (i.e.  $\neg C$  is entailed), **set**  $B$  to 0

## Detecting entailment – levels of precision

Consider a reified constraint of the form  $C \#<=> B$

- If  $C$  is a **membership** constraint, detecting **domain-entailment** is guaranteed, i.e.  $B$  is set as soon as  $C$  or  $\neg C$  is entailed by the store, **e.g.**
    - $| \text{?- } X \text{ in } 1..3, X \text{ in } \{1,3\} \#<=> B, X \# \backslash = 2. \implies B = 1, X \text{ in } \{1\} \backslash \{3\}$
    - $| \text{?- } X \text{ in } 2..4, X \text{ in } \{1,3\} \#<=> B, X \# \backslash = 3. \implies B = 0, X \text{ in } \{2\} \backslash \{4\}$
  - If  $C$  is a **linear arithmetic** constraint, detecting **bound-entailment** is guaranteed, i.e.  $B$  is set as soon as  $C$  or  $\neg C$  is entailed by the **interval closure** of the store. (Recall: The interval closure of the store maps each variable  $X$  to  $\text{Min}X.. \text{Max}X$ , where  $\text{Min}X/\text{Max}X$  is the smallest/largest value in  $X$ 's domain)
    - **Store**:  $X$  in  $\{1,3\}$ ,  $Y$  in  $\{2,4\}$ ,  $Z$  in  $\{2,4\}$
    - **Interval closure** of the store:  $X$  in  $1..3$ ,  $Y$  in  $2..4$ ,  $Z$  in  $2..4$
- E.g.**  $X$  in  $\{1,3\}$ ,  $Y$  in  $\{2,4\}$ ,  $Z$  in  $\{2,4\}$ ,  $(X+Y\# \backslash = Z) \#<=> B \implies B$  in  $0..1$   
 The **store** entails  $X+Y \neq Z$  (odd+even  $\neq$  even), but its **intv. closure** does not!
- No guarantee is given for **non-linear arithmetic** constraints, but when a constraint becomes ground, its (dis)entailment is always detected

## Detecting entailment – some further examples in SICStus

- Bound-entailment is guaranteed for linear arithmetic constraints
- However, for certain constraints you can obtain better entailment detection in SICStus Prolog
- Domain entailment is detected in an inequality between two variables:
  - $| \text{?- } X \text{ in } \{1,3,7,9\}, Y \text{ in } \{2,8,10\}, X \# \backslash = Y \#<=> B. \implies B = 1$
- Domain entailment can be obtained for linear arithmetic constraints by replacing the formula constraint by the `scalar_product/4` global constraint, with the `consistency(domain)` option

Bound entailment, using a formula constraint:

```
| ?- X in {1,3}, Y in {2,4}, Z in {2,4}, X+Y # \ = Z #<=> B.
 => B in 0..1
```

Domain entailment, using `scalar_product/4`:

```
| ?- X in {1,3}, Y in {2,4}, Z in {2,4},
 scalar_product([1,1], [X,Y], # \ =, Z, [consistency(domain)]) #<=> B.
 => B = 1
```

## Knights and knaves – a CLPFD example using Booleans

- Knights and knaves puzzle (“What is the name of this book” by R. Smullyan)
  - A remote island is inhabited by two kinds of natives:
    - knights* always tell the truth, *knaves* always lie.
    - One day I meet two natives, A and B. A says: “One of us is a knave”. What are A and B?
- Operators used in the [controlled natural language](#) syntax below:
 

```
:- op(100,fy,a), op(700,fy,not), op(800,yfx,and), op(900,yfx,or), op(950,xfy,says).
```
- Prolog representation: knave (liar)  $\rightarrow$  0, knight (truthful)  $\rightarrow$  1.
- Example runs:
 

```
| ?- holds(A says A is a knave or B is a knave).
 => A = knight, B = knave ? ; no
| ?- holds((A says B is a knight) and (B says C is a knight)).
 => A = knave, B = knave, C = knave ? ;
 A = knight, B = knight, C = knight ? ; no
```
- 0 and 1 are displayed as knave and knight via callback pred. `portray/1`:
 

```
:- multifile portray/1. % clauses for portray can be scattered over multiple files
portray(0) :- write(knave).
portray(1) :- write(knight).
```

## Knights and knaves – CLPFD solution

```
:- use_module(library(clpfd)).
:- op(100, fy, a), op(700, fy, not), op(800, yfx, and), op(900, yfx, or), op(950, xfy, says).

holds(Stmt) :-
 term_variables(Stmt, Vars),
 % term_variables(+T, -Vs): Vs is the list of vars that occur in term T
 domain(Vars, 0, 1),
 has_value(Stmt, 1), labeling([], Vars).

% native(Nat, V): The truth value of sentences spoken by native Nat is V.
native(knave, 0).
native(knight, 1).

% has_value(Stmt, Val): The truth value of statement Stmt is Val.
has_value(X is a Nat, V) :- native(Nat, N), V #<=> X #= N.
has_value(X says S, V) :- has_value(S, V0), V #<=> X #= V0.
has_value(S1 and S2, V) :- has_value(S1, V1),
 has_value(S2, V2), V #<=> V1 #/\ V2.
has_value(S1 or S2, V) :- has_value(S1, V1),
 has_value(S2, V2), V #<=> V1 #\ V2.
has_value(not S1, V) :- has_value(S1, V1), V #<=> #\ V1.
```

## Contents

- 3 Declarative Programming with Constraints
  - Motivation
  - CLPFD basics
  - How does CLPFD work
  - FDBG
  - Reified constraints
  - Global constraints**
  - Labeling
  - From plain Prolog to constraints
  - Improving efficiency
  - Internal details of CLPFD
  - Disjunctions in CLPFD
  - Modeling
  - User-defined constraints (ADVANCED)
  - Some further global constraints (ADVANCED)
  - Closing remarks

## Global constraints – an overview

| Category           | Constraint                                                                                                       |
|--------------------|------------------------------------------------------------------------------------------------------------------|
| Counting           | <code>count/4</code><br><code>global_cardinality/[2,3]</code><br><code>nvalue/2</code>                           |
| Sorting            | <code>sorting/3</code><br><code>lex_chain/[1,2]</code>                                                           |
| Distinctness       | <code>all_different/[1,2]</code><br><code>all_distinct/[1,2]</code>                                              |
| Permutation        | <code>assignment/[2,3]</code><br><code>circuit/[1,2]</code>                                                      |
| Scheduling         | <code>cumulative/[1,2]</code><br><code>cumulatives/[2,3]</code>                                                  |
| Geometric          | <code>disjoint1/[1,2]</code><br><code>disjoint2/[1,2]</code><br><code>geost/[2,3,4]</code>                       |
| Arbitrary relation | <code>automaton/[3,8,9]</code><br><code>case/[3,4]</code><br><code>relation/3</code><br><code>table/[2,3]</code> |
| Other              | <code>element/3</code>                                                                                           |

## Arguments of global constraints

- It is important to differentiate between two kinds of arguments:
  - Arguments that can be FD-variables (or lists of such)
  - Arguments that can only be integers (or lists of such)
- It is always possible to write an integer where an FD-variable is expected, but not the other way around
- Convention: in this section, FD-variables (and lists of such) are written in *italics*.

Simple counting: `count/4`

- `count/4` can be used to count the occurrences of a given integer, e.g. `count(0, L, #=, N)`.  $\equiv$  there are exactly  $N$  zero elements in  $L$ .
- `count(Int, List, RelOp, Count)`:  $\text{Int}$  occurs in  $List$   $n$  times, and  $(n \text{ RelOp } Count)$  holds. (Not available in SWI-Prolog)
 

```
| ?- length(L, 3), % L is a list of 3 elements
 domain(L, 6, 8), % all elements of L are between 6 and 8
 count(7, L, #=, 3). % There are exactly 3 occurrences of 7 in L
 => L = [7,7,7] ? ; no

| ?- length(L, 3), domain(L, 1, 100),
 count(3, L, #=, _C),
 _C #>= 1, % There is at least one 3 in L
 count(2, L, #>, _C), % There are more 2's than 3's in L
 labeling([], L).
 => L = [2,2,3] ? ; L = [2,3,2] ? ; L = [3,2,2] ? ; no
```
- `count` can be implemented using reification (this works in SWI):
 

```
count(Val, List, RelOp, Count) :- maplist(count1(Val), List, Bs),
 sum(Bs, RelOp, Count).

count1(Val, X, B) :- X #= Val #<==> B.
```

Counting multiple values: `global_cardinality/2`

- This constraint can be used to describe the exact composition of a list.
- E.g.,  $L$  contains ints 0, 1, and 2 only, the count of 1's and 2's is the same:
 

```
| ?- L=[_,_], global_cardinality(L, [0-C0,1-C,2-C]), labeling([], L).
 L = [0,0], C0 = 2, C = 0 ? ;
 L = [1,2], C0 = 0, C = 1 ? ;
 L = [2,1], C0 = 0, C = 1 ? ; no
```
- The definition of `global_cardinality(Vars, [K1-V1, ...Kn-Vn])`:
  - $K_1, \dots, K_n$  are distinct integers,
  - each of the  $Vars$  takes a value from  $\{K_1, \dots, K_n\}$ ,
  - each integer  $K_i$  occurs exactly  $V_i$  times in  $Vars$ , for all  $1 \leq i \leq n$ .

```
| ?- length(L, 3), global_cardinality(L, [6-,7-3,8-]).
 L = [7,7,7] ? ; no

| ?- length(L,3), domain(L,1,100), global_cardinality(L,[2-_X,3-_Y]),
 _X#>_Y, _Y#>0, labeling([], L).
 => L = [2,2,3] ? ; L = [2,3,2] ? ; L = [3,2,2] ? ; no
```
- There is a variant `global_cardinality/3` with a 3rd, `Options` argument, where pruning strength can be specified

## Distinctness

- `all_distinct(Vars, Options)`  
`all_different(Vars, Options)`: Variables in  $Vars$  are pairwise different. The two predicates differ only in `Options` defaults. An empty `Options` argument can be omitted.
 

```
| ?- L = [A,B,C], domain(L,1,2), all_different(L). => A in 1..2, ...
| ?- L = [A,B,C], domain(L,1,2), all_distinct(L). => no
```
- The `Options` argument is a list of options. In the option `consistency(Cons)`, `Cons` controls the strength of the pruning:
  - `Cons = domain` (the default for `all_distinct`): strongest possible pruning (domain consistency)
  - `Cons = value` (the default for `all_different`): strength equivalent to posting `#\=` for all variable pairs
  - `Cons = bounds`: bounds consistency
- In SICStus other options are also available
- SWI-Prolog only supports the 1-argument version (no options argument)

## Permutation (ADVANCED)

- assignment( $[X_1, \dots, X_n], [Y_1, \dots, Y_n]$ ): all  $X_i, Y_i$  are in  $1..n$  and  $X_i=j$  iff  $Y_j=i$ . Equivalently:  $[X_1, \dots, X_n]$  is a permutation of  $1..n$  and  $[Y_1, \dots, Y_n]$  is the inverse permutation.

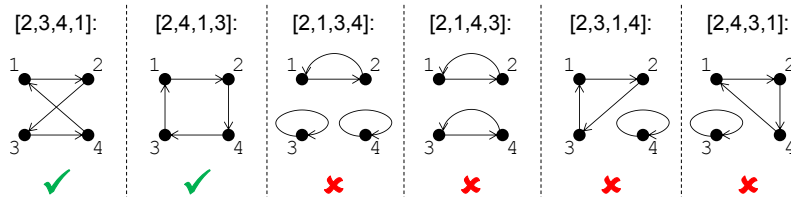
```
| ?- length(Xs, 3), assignment(Xs, Ys), Ys = [3|_], labeling([], Xs).
 => Xs = [2,3,1], Ys = [3,1,2] ? ;
 => Xs = [3,2,1], Ys = [3,2,1] ? ; no
```

- circuit( $[X_1, \dots, X_n]$ ):

Edges  $i \rightarrow X_i$  form a single (Hamiltonian) circuit of nodes  $\{1, \dots, n\}$ .

Equivalently:  $[X_1, \dots, X_n]$  is a permutation of  $1..n$  that consists of a single cycle of length  $n$ .

```
| ?- length(Xs, 4), circuit(Xs), Xs = [2|_], labeling([], Xs).
 => Xs = [2,3,4,1] ? ;
 => Xs = [2,4,1,3] ? ; no
```



## Specifying arbitrary finite relations

- table( $[Tuple1, \dots, TupleN], Extension$ ): each  $Tuple$  belongs to the relation described by  $Extension$ .  $Extension$  is a list of all the valid tuples that form the relation. Available in SWI-Prolog as `tuples_in/2`.

```
% times(X, Y, Z): X * Y = Z, for 1 <= X, Y <= 4
times(X, Y, Z) :- table([[X,Y,Z]], [[1,1,1], [1,2,2], [1,3,3], [1,4,4],
 [2,1,2], [2,2,4], [2,3,6], [2,4,8],
 [3,1,3], [3,2,6], [3,3,9], [3,4,12],
 [4,1,4], [4,2,8], [4,3,12], [4,4,16]]).
```

```
| ?- times(X, 4, Z), Z #> 10. => X in 3..4, Z in {12}\{16} ? ; no
```

- If the 1st arg. contains several tuples, each has to belong to the relation.

Example: find paths  $x$ - $Y$ - $Z$  in the graph  $\{1 \rightarrow 3, 4 \rightarrow 6, 3 \rightarrow 5, 6 \rightarrow 8\}$

```
| ?- table([[X,Y],[Y,Z]], [[1,3],[4,6],[3,5],[6,8]]), labeling([], [X,Y,Z]).
X = 1, Y = 3, Z = 5 ? ; X = 4, Y = 6, Z = 8 ? ; no
```

- table/2 produces the same solutions as a collection of member/2 goals:

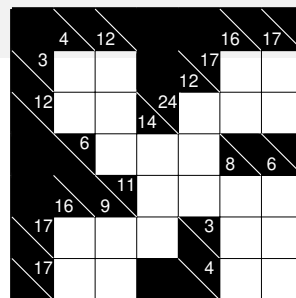
```
| ?- Ext = [[1,3],[4,6],[3,5],[6,8]], member([X,Y], Ext), member([Y,Z], Ext).
X = 1, Y = 3, Z = 5 ? ; X = 4, Y = 6, Z = 8 ? ; no
```

- table/2 provides domain consistency:

```
| ?- table([[X,Y],[Y,Z]], [[1,3],[4,6],[3,5],[6,8]]).
X in {1}\{4}, Y in {3}\{6}, Z in {5}\{8} ?
```

## Specifying arbitrary finite relations, cntd.

- A kakuro puzzle – a crossword using digits instead of letters:
- Each sequence (across or down)
  - contains **different** digits
  - sums** to the number given as a clue



- table/2 can be used for combining these **two constraints**, to make the search more efficient:

```
% List L, containing integers between 1 and N, sums to Sum.
diffsum(L, N, Sum) :-
 domain(L, 1, N), % all elements of L are between 1 and N
 append(L, [Sum], L1),
 findall(L1, (sum(L, #=, Sum), all_different(L), labeling([], L)),
 Tuples),
 table([L1], Tuples).
| ?- length(L, 3), diffsum(L, 9, 24).
=> L = [_A,_B,_C], _A in 7..9, _B in 7..9, _C in 7..9 ?
```

- Using `diffsum`, the above puzzle can be solved without labeling.

## Getting an element of a list

- element( $X, List, Y$ ):  
 $Y$  is the  $X^{th}$  element of  $List$  (counting from 1)
- element/3 is the FD counterpart of the predicate `nth1/3`, `library(lists)`
- Examples:

```
| ?- L=[A,B,C], domain(L, 1, 5),
 B#<3, Y in 4..6,
 element(X, L, Y).
=> ..., X in {1}\{3}, Y in 4..5 ?
```

*% domain-consistent in X: only the 1st and 3rd elements belong to 4..5*

```
| ?- L = [A,B], A in 1..2, B in 5..7,
 element(X, L, Y).
=> ..., X in 1..2, Y in 1..7 ?
```

*% only bound-consistent in Y, as the exact domain is (1..2)\(5..7)*

## Class practice task

Write a constraint (predicate) according to the spec below

- Partitioning a list

```
% partition(+L1, ?L2): L1 is a list of integers; L2 contains a subset of
% the elements of L1 (in the same order as in L1), such that the sum of
% elements in L2 is half of the sum of elements in L1.
```

```
| ?- partition([1,2,3,5,8,13], L2).
L2 = [3,13] ? ;
L2 = [3,5,8] ? ;
L2 = [1,2,13] ? ;
L2 = [1,2,5,8] ? ; no
```

Hint: it is helpful to use  $n$  binary variables (where  $n$  denotes the number of elements of `L1`), with  $x_i = 1$  meaning that the  $i$ th element of `L1` should also be an element of `L2` and  $x_i = 0$  otherwise. It is fairly easy to formulate the constraint in terms of these variables. After labeling, do not forget to create the desired output based on the values of the  $x_i$  variables.

## Contents

### 3 Declarative Programming with Constraints

- Motivation
- CLPFD basics
- How does CLPFD work
- FDBG
- Reified constraints
- Global constraints
- Labeling**
- From plain Prolog to constraints
- Improving efficiency
- Internal details of CLPFD
- Disjunctions in CLPFD
- Modeling
- User-defined constraints (ADVANCED)
- Some further global constraints (ADVANCED)
- Closing remarks

## Labeling – recap

- Typical CLPFD program structure:
  - Define variables and domains
  - Post constraints (no choice points!)
  - Labeling**
  - Optional post-processing
- Labeling traverses the search tree – the search space of possible variable assignments – using a depth-first strategy (cf. Prolog execution)
- Labeling creates choice points (decision points), manages all the branching and backtracking
- Each decision is normally followed by **propagation**: constraints wake up, perform pruning, further constraints may wake up etc.

## Labeling – overview

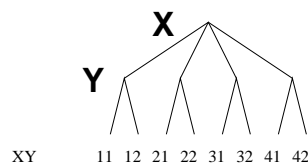
- Possible aims of labeling:
  - Find a single solution (decide solvability)
  - Find all solutions
  - Find the best solution according to a given objective function (not covered in detail)
- In general, labeling guarantees a *complete* search, i.e. all solutions are enumerated (advanced options, e.g. `timeout` may cause incompleteness)
- A typical CLPFD program spends almost 100% of its running time in the call to `labeling`  $\implies$  efficiency is critical
- Efficiency largely depends on the main **search options**:
  - How to choose a variable to branch on
  - Way of splitting the domain of the chosen variable
  - Order of considering the possible values of the chosen variable



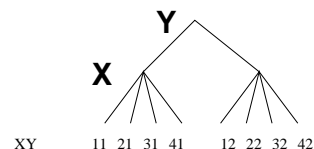
## Order of the variables to branch on

- `| ?- X in 1..4, Y in 1..2, XY #= 10*X+Y, indomain(X), indomain(Y).`

`indomain(X)` creates a choice point enumerating all possible values for `x`



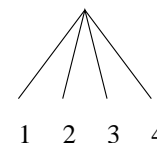
- `| ?- X in 1..4, Y in 1..2, XY #= 10*X+Y, indomain(Y), indomain(X).`



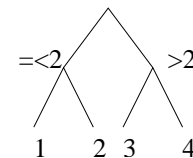
- The order of the variables can have significant impact on the number of visited tree nodes
- **First-fail** principle: start with the variable that has the smallest domain
- **Most-constrained** principle: start with the variable that has the most constraints suspended on it

## How to split the domain of the selected variable?

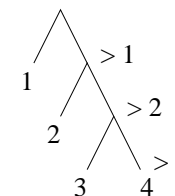
- enumeration: `| ?- X in 1..4, labeling([enum], [X]).`



- bisection: `| ?- X in 1..4, labeling([bisect], [X]).`



- stepping: `| ?- X in 1..4, labeling([step], [X]).`



## Labeling predicates

`labeling(Options, VarList):`

- Enumerates all possible value assignments of the variables in `VarList`
- All vars in `VarList` must have **finite domains**, otherwise an error is raised
- The `Options` argument may contain at most one from each of the following **option categories** (default values are in *italics*, options shown in **brown** are available only in SICStus, and are not discussed in detail)
  - **Variable selection:** *leftmost*, *min*, *max*, *ff*, *ffc*, ..., *anti\_first\_fail*, *occurrence*, *max\_regret*, *variable(Sel)*
  - **Type of splitting:** *step*, *enum*, *bisect*, ..., *value(Enum)*
  - **Order of children:** *up*, *down*, ..., *median*, *middle*
  - **Objective:** *satisfy*, ..., *minimize(Var)*, *maximize(Var)*
  - **Time limit:** *time\_out(RunTimeInMSec,Result)*

`indomain(X):` is equivalent to `labeling([enum], [X]).`

## Options for variable selection

- **leftmost** (default) — use the order as the variables were listed
- **min** — choose the variable with the smallest lower bound
- **max** — choose the variable with the highest upper bound
- **ff** — ('first-fail' principle): choose the variable with the smallest domain
- **occurrence** — ('most-constrained' principle): choose the variable that has the most constraints suspended on it
- **ffc** — (combination of 'first-fail' and 'most-constrained' principles): choose the variable with the smallest domain; if there is a tie, choose the variable that has the most constraints suspended on it
- **anti\_first\_fail** — choose the variable with the largest domain
- ...

For tie-breaking, `leftmost` is used

## Options for branching

### Type of splitting:

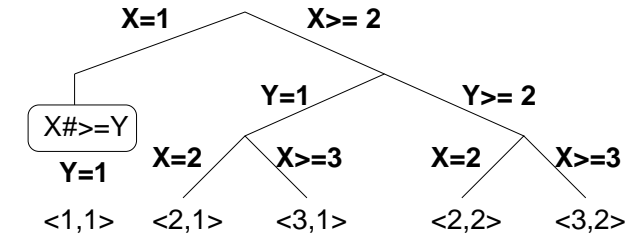
- `step` (default) — two-way branching according to  $X \# = LB$  vs.  $X \# \setminus = LB$ , where  $LB$  is the lower bound of the domain of  $X$ ; or — if option `down` applies, see below — according to  $X \# = UB$  vs.  $X \# \setminus = UB$ , (upper bound)
- `enum` —  $n$ -way branching, enumerating all  $n$  possible values of  $X$
- `bisect` — two way branching according to  $X \# \leq M$  vs.  $X \# > M$ , where  $M$  is the middle of the domain of  $X$  ( $M = (\min(X) + \max(X)) // 2$ )
- ...

### Direction:

- `up` (default) — the domain is enumerated in ascending order
- `down` — the domain is enumerated in descending order
- ...

## Labeling – a simple example

- Sample query:  
 $X$  in  $1..3$ ,  $Y$  in  $1..2$ ,  $X \# \geq Y$ , `labeling`(`[min]`, `[X,Y]`).
- Option `min` means: select the variable that has the smallest lower bound
  - If there is a tie, select the leftmost
- No option provided for branching  $\implies$  defaults used (`step` and `up`)
- The search tree:



## Impact on performance

Time for finding all solutions of  $N$ -queens for  $N = 13$   
(on an Intel i5-3230M 2.60GHz CPU):

| Labeling options               | Runtime    |
|--------------------------------|------------|
| <code>[leftmost,step]</code>   | 6.295 sec  |
| <code>[leftmost,enum]</code>   | 5.604 sec  |
| <code>[leftmost,bisect]</code> | 6.281 sec  |
| <code>[min,step]</code>        | 6.610 sec  |
| <code>[min,enum]</code>        | 6.633 sec  |
| <code>[min,bisect]</code>      | 12.081 sec |
| <code>[ff,step]</code>         | 5.134 sec  |
| <code>[ff,enum]</code>         | 4.716 sec  |
| <code>[ff,bisect]</code>       | 5.180 sec  |
| <code>[ffc,step]</code>        | 5.264 sec  |
| <code>[ffc,enum]</code>        | 4.854 sec  |
| <code>[ffc,bisect]</code>      | 5.214 sec  |

## Contents

- Declarative Programming with Constraints
  - Motivation
  - CLPFD basics
  - How does CLPFD work
  - FDBG
  - Reified constraints
  - Global constraints
  - Labeling
  - From plain Prolog to constraints
  - Improving efficiency
  - Internal details of CLPFD
  - Disjunctions in CLPFD
  - Modeling
  - User-defined constraints (ADVANCED)
  - Some further global constraints (ADVANCED)
  - Closing remarks

## Transforming Prolog code to constraint code – an example

```
% pcountVT(L, N): L has N positive elements.
% Predicate naming convention:
% V = <single digit> version number
% T = p | c for plain Prolog vs. CLPFD
```

Step 1: ensure there is a single recursive call within the predicate

```
pcount0p([], 0).
pcount0p([X|Xs], N) :-
 (X > 0 ->
 pcount0p(Xs, NO),
 N is NO+1
 ; pcount0p(Xs, N)
).

pcount1p([], 0).
pcount1p([X|Xs], N) :-
 pcount1p(Xs, NO),
 (X > 0 ->
 N is NO+1
 ; N = NO
).
```

Note that the if-then-else contains arithmetic and equality BIPs only. This is important when transforming to CLPFD.

## Prolog to constraints – a simple example, ctd.

A scheme to convert Prolog if-then-else to CLPFD code using reification:

```
foo(...) :- NonrecTest.
foo(...) :-
 foo(...),
 (Cond -> Then
 ; Else
).

foo(...) :- NonrecTest#.
foo(...) :-
 foo(...),
 Cond# #<=> B,
 B #=> Then#,
 #\ B #=> Else#.
```

Step2: apply the above scheme to the Prolog predicate obtained in step 1:

```
pcount1p([], 0).
pcount1p([X|Xs], N) :-
 pcount1p(Xs, NO),
 (X > 0 -> N is NO+1
 ; N = NO
).

pcount2c([], 0).
pcount2c([X|Xs], N) :-
 pcount2c(Xs, NO),
 X #> 0 #<=> B,
 B #=> N #= NO+1,
 #\ B #=> N #= NO.
```

Note that pcount2c can be made tail recursive by simply reordering goals.

## Prolog to constraints – a simple example, cont'd.

Notice that pcount2c has bad pruning behavior:

```
| ?- pcount2c([A,B], N).
(...) N in inf..sup ? % N could be pruned to 0..2
| ?- pcount2c([A,B], N), A #> 4.
(...) N in inf..sup ? % N could be pruned to 1..2
```

Exactly one LHS of these two implications is bound to be true:

```
B #=> N #= NO+1, % if B=1, N is 1 bigger than NO
#\ B #=> N #= NO. % if B=0, N is the same as NO
```

but Prolog is not aware of this. To make Prolog able to reason, replace these two constraints by an equivalent constraint  $N \# = NO+B$ .

Prolog is now aware that N is either equal to or 1 larger than variable NO!

```
pcount3c([], 0).
pcount3c([X|Xs], N) :-
 X #> 0 #<=> B, N #= NO+B, pcount3c(Xs, NO).
```

```
| ?- pcount3c([A,B], N), A #> 4. => N in 1..2
```

## Prolog to constraints – another example – X-Sums Sudoku.

**X-Sums Sudoku**

|    |    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|
|    | 44 | 1  | 7  | 32 | 13 | 36 | 45 | 24 | 12 |    |
| 42 |    |    |    |    |    |    |    |    |    | 18 |
| 45 |    |    |    |    |    |    |    |    |    | 21 |
| 25 |    |    |    |    |    |    |    |    |    | 20 |
| 40 |    |    |    |    |    |    |    |    |    | 5  |
| 32 |    |    |    |    |    |    |    |    |    | 30 |
| 21 |    |    |    |    |    |    |    |    |    | 45 |
| 10 |    |    |    |    |    |    |    |    |    | 1  |
| 14 |    |    |    |    |    |    |    |    |    | 42 |
| 1  |    |    |    |    |    |    |    |    |    | 33 |
|    | 1  | 41 | 20 | 3  | 41 | 26 | 9  | 45 | 33 |    |

Rajesh Kumar @ www.FunWithPuzzles.com

Basic Sudoku rules apply. Additionally the clues outside the grid indicate the sum of the first X numbers placed in the corresponding direction, where X is equal to the first number placed in that direction.

This requires the following constraint:

```
nsum(L, N, Sum): The first N elements of list L add up to Sum.
```

## The `nsum` constraint

- We follow the same steps as for `pcount`
- Common specification:  

```
% nsumVT(Xs, N, Sum): The leftmost N elements of Xs add up to Sum.
```
- First Prolog version:  

```
nsum0p([], 0, 0).
nsum0p([X|Xs], NO, Sum0) :-
 (NO > 0
 -> N1 is NO-1, Sum1 is Sum0-X, nsum0p(Xs, N1, Sum1)
 ; Sum0 = 0
).
```
- We have an additional problem here: this recursion stops when `NO` becomes 0. However, in the constraint version `NO` may not be known yet.
- Solution: we transform this code so that it always scans the whole list. (This is an unnecessary overhead in the Prolog version, but is needed for the constraint version.)

## The `nsum` constraint, cont'd.

- Second Prolog version:  

```
nsum1p([], 0, 0).
nsum1p([X|Xs], NO, Sum0) :-
 (NO > 0 -> N1 is NO-1, Sum1 is Sum0-X
 ; N1 = NO, Sum1 = Sum0
),
 nsum1p(Xs, N1, Sum1).
```
- Notice that when the counter `NO` becomes 0 we keep the recursion running, without changing the sum and the counter.
- The two CLPFD versions:

```
nsum2c([], 0, 0).
nsum2c([X|Xs], NO, Sum0) :-
 NO #> 0 #<=> B,
 B #=> N1 #= NO-1 #/\ Sum1 #= Sum0-X,
 #\ B #=> N1 #= NO #/\ Sum1 #= Sum0,
 nsum2c(Xs, N1, Sum1).

nsum3c([], 0, 0).
nsum3c([X|Xs], NO, Sum0) :-
 NO #> 0 #<=> B,
 N1 #= NO-B,
 Sum1 #= Sum0-X*B,
 nsum3c(Xs, N1, Sum1).
```

## Contents

- 3 Declarative Programming with Constraints
  - Motivation
  - CLPFD basics
  - How does CLPFD work
  - FDBG
  - Reified constraints
  - Global constraints
  - Labeling
  - From plain Prolog to constraints
  - Improving efficiency
  - Internal details of CLPFD
  - Disjunctions in CLPFD
  - Modeling
  - User-defined constraints (ADVANCED)
  - Some further global constraints (ADVANCED)
  - Closing remarks

## Techniques for improving efficiency of CLPFD programs

In most cases:

- Avoiding choice points (other than `labeling`)
- Finding the most appropriate labeling options

In some cases:

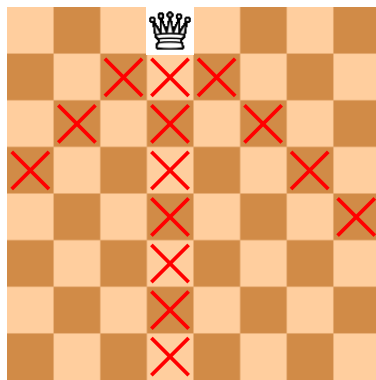
- Reordering the variables before labeling
- Introducing symmetry breaking rules to exclude equivalent solutions
- Using global constraints instead of several 'small' constraints
- Using redundant constraints for additional pruning
- Using constructive disjunction and shaving to prune infeasible values
- Trying alternative models of the problem

Further options (not discussed in detail):

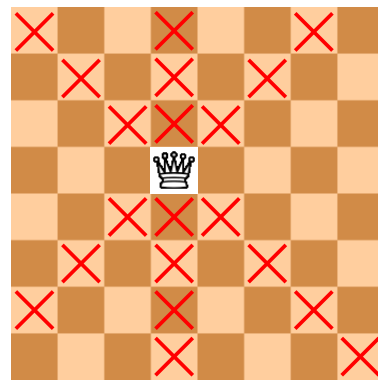
- Custom labeling heuristics
- Experimenting with the possible options of library constraints
- Implementing user-defined constraints with improved pruning capabilities

## Reordering the variables before labeling

Example: in the  $N$ -queens problem, how many values can be pruned from the domain of other variables, after instantiating a variable?



⇒ 14



⇒ 20

Idea: variables should be instantiated inside-out, starting from the middle

## Reordering the variables before labeling

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

```
% reorder_inside_out(+L1, -L2): L2 contains the same elements as L1
% but reordered inside-out, starting from the middle, going alternately
% up and down
```

```
reorder_inside_out(L1, L2) :-
 length(L1,N),
 Half1 is N//2, Half2 is N-Half1,
 prefix_length(L1,FirstList,Half1), suffix_length(L1,SecondList,Half2),
 reverse(FirstList,ReversedFirstList),
 merge(ReversedFirstList,SecondList,L2).
```

```
% merge(+L1, +L2, -L3): the elements of L3 are alternately the
% elements of L1 and L2.
```

```
merge([], [], []).
merge([X], [], [X]).
merge([], [Y], [Y]).
merge([X|L1], [Y|L2], [X,Y|L3]) :-
 merge(L1,L2,L3).
```

## Reordering the variables before labeling

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

```
% queens_clpfd(N, Qs): Qs is a valid placement of N queens on an NxN
% chessboard.
```

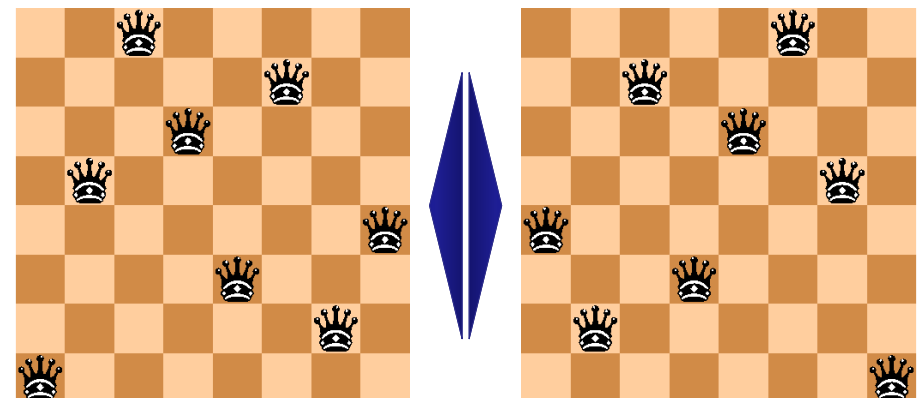
```
queens_clpfd(N, Qs) :-
 placement(N, N, Qs),
 safe(Qs),
 reorder_inside_out(Qs,Qs2),
 labeling([ffc,bisect],Qs2).
```

⇒ Time in msec for finding all solutions of  $N$ -queens for  $N = 12$  (on an Intel i3-3110M, 2.40GHz CPU):

| Without reordering | With reordering |
|--------------------|-----------------|
| 1,810              | 1,311           |

## Symmetry breaking

- Symmetry: a solution induces other – in a sense, equivalent – solutions
- Symmetry breaking: narrowing the search space by eliminating some of the equivalent solutions
- Example:  $N$ -queens – mirrored solutions



## Symmetry breaking

- A simple symmetry-breaking rule for  $N$ -queens: the queen in the first row must be in the left half of the row  
Mid is  $(N+1)//2$ ,  $Qs=[Q1|_]$ ,  $Q1\#\leq$ Mid
- This will roughly halve the runtime
- Only half of the solutions will be found
- If all solutions are needed, the remaining ones must be created by mirroring

## Another case study: magic sequences

- **Definition:**  $L = (x_0, \dots, x_{n-1})$  is a *magic sequence* if
  - each  $x_i$  is an integer from  $[0, n-1]$  and
  - for each  $i = 0, 1, \dots, n-1$ , the number  $i$  occurs exactly  $x_i$  times in  $L$
- **Examples** for  $n = 4$ :  $(1, 2, 1, 0)$  and  $(2, 0, 2, 0)$
- **Problem:** write a CLPFD program that finds a magic sequence of a given length, and enumerates all solutions on backtracking  
% magic(+N, ?L): L is a magic sequence of length N.

## Solution, main part

```
% magic(+N, ?L): L is a magic sequence of length N.
magic(N,L) :-
 length(L,N),
 N1 is N-1, domain(L,0,N1),
 occurrences(L,0,L),
 labeling([ffc],L).

% occurrences(Suffix, I, L): Suffix is the suffix of L starting at
% position I, and the magic sequence constraint holds for each element of
% Suffix.
occurrences([],_,_).
occurrences([X|Suffix],I,L) :-
 exactly(I,L,X),
 I1 is I+1,
 occurrences(Suffix,I1,L).

% exactly(I,L,X): the number I occurs exactly X times in list L.
```

## Variations for exactly/3

```
% exactly(I,L,X): the number I occurs exactly X times in list L.
```

- **Speculative** solution (uses choice points in posting the constraints):
 

```
exactly_spec(I, [I|L], X) :- % next element is I
 X#>0, X1 #= X-1, exactly_spec(I, L, X1).
exactly_spec(I, [J|L], X) :- % I is expected later
 X#>0, J #\= I, exactly_spec(I, L, X).
exactly_spec(I, L, 0) :- % no I left in list
 maplist(#\=(I), L).
```
- A non-speculative solution using **reification**:
 

```
exactly_reif(_, [], 0).
exactly_reif(I, [J|L], X) :-
 J#=I #<=> B, X#=X1+B,
 exactly_reif(I, L, X1).
```

Variations for `exactly/3`, cntd.

- A non-speculative solution using a `global` library constraint:

```
exactly_glob(I, L, X) :-
 count(I, L, #=, X).
```

- A single `global_cardinality/2` library constraint:

```
magic(N,L) :-
 length(L,N),
 N1 is N-1, domain(L,0,N1),
 enumerate(0, L, Pairs),
 global_cardinality(L, Pairs),
 labeling([ffc],L).

enumerate(_, [], []).
enumerate(N, [Elem|Rest], [N-Elem|PairRest]) :-
 N1 is N + 1,
 enumerate(N1, Rest, PairRest).
```

## Evaluation

Time for all solutions in msec:

| N  | Speculative | Reification | Global | Global card. |
|----|-------------|-------------|--------|--------------|
| 6  | 10          | 20          | 0      | 10           |
| 9  | 280         | 20          | 0      | 10           |
| 10 | 1,190       | 20          | 10     | 0            |
| 12 | 34,040      | 10          | 0      | 0            |
| 13 | >180,000    | 10          | 10     | 0            |
| 20 |             | 70          | 30     | 0            |
| 40 |             | 1,150       | 340    | 0            |

## Redundant constraints

- **Proposition 1:** If  $L = (x_0, \dots, x_{n-1})$  is a magic sequence, then

$$\sum_{i=0}^{n-1} x_i = n$$

- Implementation using CLPFD:

```
sum(L, #=, N)
```

- **Proposition 2:** If  $L = (x_0, \dots, x_{n-1})$  is a magic sequence, then

$$\sum_{i=0}^{n-1} i \cdot x_i = n$$

- Implementation using CLPFD (using also `library(between)`):

```
N1 is N-1,
numlist(0, N1, Coeffs), % Coeffs = [0,1,...,N1]
scalar_product(Coeffs, L, #=, N)
```

## The effect of redundant constraints on the global approach

Time in msec:

- P = proposition

| N    | count  |       |       |       | global_cardinality |       |       |       |
|------|--------|-------|-------|-------|--------------------|-------|-------|-------|
|      | None   | P1    | P2    | P1+P2 | None               | P1    | P2    | P1+P2 |
| 80   | 4,950  | 230   | 30    | 30    | 10                 | 10    | 10    | 10    |
| 160  | 83,960 | 1,970 | 190   | 200   | 30                 | 20    | 30    | 30    |
| 320  |        | 33.2s | 1.6s  | 1.5s  | 140                | 100   | 110   | 120   |
| 480  |        | 145s  | 7.0s  | 6.8s  | 240                | 240   | 240   | 260   |
| 640  |        | >200s | 16.4s | 16.4s | 470                | 440   | 450   | 500   |
| 800  |        |       | 31.4s | 31.4s | 740                | 730   | 730   | 720   |
| 1600 |        |       |       |       | 3.7s               | 3.8s  | 3.8s  | 4.1s  |
| 3200 |        |       |       |       | 18.4s              | 19.7s | 18.9s | 19.9s |

## Contents

- 3 Declarative Programming with Constraints
  - Motivation
  - CLPFD basics
  - How does CLPFD work
  - FDBG
  - Reified constraints
  - Global constraints
  - Labeling
  - From plain Prolog to constraints
  - Improving efficiency
  - Internal details of CLPFD
  - Disjunctions in CLPFD
  - Modeling
  - User-defined constraints (ADVANCED)
  - Some further global constraints (ADVANCED)
  - Closing remarks

## FD variable internals – reflection predicates

(The slides in this section are specific to SICStus Prolog)

- CLPFD stores for each finite domain (FD) variable:
  - the size of the domain
  - the lower bound of the domain
  - the upper bound of the domain
  - the domain as an FD-set (internal representation format)
- The above pieces of information can be obtained (in constant time) using
  - `fd_size(X, Size)`: `Size` is the size (number of elements) of the domain of `x` (integer or `sup`).
  - `fd_min(X, Min)`: `Min` is the lower bound of `x`'s domain; `Min` can be an integer or the atom `inf`
  - `fd_max(X, Max)`: `Max` is the upper bound of `x`'s domain (integer or `sup`).
  - `fd_set(X, Set)`: `Set` is the domain of `x` in FD-set format
  - `fd_degree(X, D)`: `D` is the number of constraints attached to `x`
- Further reflection predicate
  - `fd_dom(X, Range)`: `Range` is the domain of `x` in *ConstRange* format (the format accepted by the constraint `Y` in *ConstRange*)

## FD reflection predicates – examples

```
| ?- X in (1..5)\/{9}, fd_min(X, Min), fd_max(X, Max),
 fd_size(X, Size).
 Min = 1, Max = 9, Size = 6, X in(1..5)\/{9} ?
```

```
| ?- X in (1..9)/\ \ (6..8), fd_dom(X, Dom), fd_set(X, Set).
 Dom = (1..5)\/{9}, Set = [[1|5],[9|9]], X in ... ?
```

To illustrate `fd_degree` here is a variant of N-queens without labeling:

```
% queens_nolab(N, Qs): Qs is a valid placement of N queens on
% an NxN chessboard. queens_nolab/2 does not perform labeling.
```

```
queens_nolab(N, Qs):-
 length(Qs, N), domain(Qs, 1, N), safe(Qs).
```

```
| ?- queens_nolab(8, [X|_]), fd_degree(X, Deg).
 Deg = 21, X in 1..8 ? % 21 = 7*3
```

## FD variable internals

- ConstRange vs. FD-set format

```
| ?- X in 1..9, X#\=5, fd_dom(X,R), fd_set(X,S).
```

```
⇒ R = (1..4)\/(6..9), S = [[1|4],[6|9]]
```

**FD-set** is an internal format; user code should not make any assumptions about it – use access predicates instead, see next slide

- When do we need access to data associated with FD variables?
  - when implementing a user-defined labeling procedure
  - when implementing a user-defined constraint
  - for other special techniques, such as **constructive disjunction** or **shaving**
- To perform the above tasks efficiently, we need predicates for processing FD-sets



## Manipulating FD-sets

Some of the many useful operations:

- `is_fdset(Set)`: Set is a proper FD-set.
- `empty_fdset(Set)`: Set is an empty FD-set.
- `fdset_parts(Set, Min, Max, Rest)`: Set consists of an initial interval `Min..Max` and a remaining FD-set `Rest`.
- `fdset_interval(Set, Min, Max)`: Set represents the interval `Min..Max`.
- `fdset_union(Set1, Set2, Union)`: The union of `Set1` and `Set2` is `Union`.
- `fdset_union(Sets, Union)`: The union of the list of FD-sets `Sets` is `Union`.
- `fdset_intersection/[2,3]`: analogous to `fdset_union/[2,3]`
- `fdset_complement(Set1, Set2)`: `Set2` is the complement of `Set1`.
- `list_to_fdset(List, Set)`, `fdset_to_list(Set, List)`: conversions between FD-sets and lists
- `X in_set Set`: Similar to `X in Range` but for FD-sets

Blue preds work back and forth, e.g. `fdset_parts(+,-,-,-)` decomposes an FD-set, while `fdset_parts(-,+,+,+)` builds an FD-set,

## Contents

- Declarative Programming with Constraints
  - Motivation
  - CLPFD basics
  - How does CLPFD work
  - FDBG
  - Reified constraints
  - Global constraints
  - Labeling
  - From plain Prolog to constraints
  - Improving efficiency
  - Internal details of CLPFD
  - Disjunctions in CLPFD
  - Modeling
  - User-defined constraints (ADVANCED)
  - Some further global constraints (ADVANCED)
  - Closing remarks

## Handling disjunctions

- Example: scheduling two tasks, both take 5 units of time
  - intervals  $[x, x + 5)$  and  $[y, y + 5)$  are disjoint
  - $(x + 5 \leq y) \vee (y + 5 \leq x)$
- Reification-based solution
 

```
| ?- domain([X,Y], 0, 6), X+5 #=< Y #\| Y+5 #=< X.
 => X in 0..6, Y in 0..6
```

no pruning
- Speculative solution
 

```
| ?- domain([X,Y], 0, 6), (X+5 #=< Y ; Y+5 #=< X).
 => X in 0..1, Y in 5..6 ? ;
 => X in 5..6, Y in 0..1 ? ; no
```

max. pruning, but choice points created
- A solution using domain-consistent arithmetic:
 

```
| ?- domain([X,Y], 0, 6),
 scalar_product([1,-1], [X,Y], #=, D, [consistency(domain)]),
 abs(D) #>= 5.
 => X in (0..1)\/(5..6), Y in (0..1)\/(5..6) ?
```

max. pruning

## Bent triples (Y-wings) – a sudoku solving technique

- Consider the following sudoku solution state, using pencilmarks (pencilmarks correspond to CLPFD variable domains)

|   | 1 | 2 | 3  | 4 | 5   | 6   |  |  |  |
|---|---|---|----|---|-----|-----|--|--|--|
| 1 |   |   | 67 |   | 126 | 236 |  |  |  |
| 2 |   |   |    |   |     | 456 |  |  |  |
| 3 |   |   | 78 |   |     | 68  |  |  |  |

- The three framed cells form a bent triple or Y-wing.
- The blue cell in r3c3 (call it  $x$ ) has two possible values: 7 and 8.
- What happens to the orange cell in r1c6 (call it  $z$ ) if  $x$  gets instantiated?
  - If  $x=7$  r1c3 becomes 6 and so 6 gets removed from the cell  $z$
  - If  $x=8$  r3c6 becomes 6 and so 6 gets removed from the cell  $z$
 Either way  $z$  cannot be 6, so we can remove 6 from  $z$
- Can 6 be removed from r1c5? And from r2c6?
- This type of reasoning is called *constructive disjunction*.

## Constructive disjunction (CD)

- Constructive disjunction is a **case-based** reasoning technique
- Assume a disjunction  $C_1 \vee \dots \vee C_n$
- Let  $D(X, S)$  denote the domain of  $X$  in store  $S$
- The idea of constructive disjunction:
  - For each  $i$ , let  $S_i$  be the store obtained by executing  $C_i$  in  $S$
  - Proceed with store  $S_U$ , the union of  $S_i$ , i.e. for all  $X$ ,  
 $D(X, S_U) = \cup_i D(X, S_i)$
- Algorithmically:
  - For each  $i$ :
    - post  $C_i$
    - save the new domains of the variables
    - undo  $C_i$
  - Narrow the domain of each variable to the union of its saved domains

## Implementing constructive disjunction (CD)

- Computing the CD of a list of constraints  $C_s$  w.r.t. a *single* variable  $Var$ :

```
cdisj(Cs, Var) :-
 findall(S, (member(C,Cs),C,fd_set(Var,S)), Doms),
 fdset_union(Doms,Set),
 Var in_set Set.
```

- Example:

```
| ?- domain([X,Y],0,6), cdisj([X+5#=<Y,Y+5#=<X], X).
 => X in(0..1)\/(5..6), Y in 0..6 ?
```

- Note that CD is not a constraint, but a one-off pruning technique.

## Shaving – a special case of constructive disjunction

- Basic idea: “What if”  $X = v$ ? (... and hope for failure). If executing  $X = v$  causes failure (without any labeling)  $\implies X \neq v$ , otherwise do nothing.

- Shaving an integer  $v$  off the domain of  $x$ :

```
shave_value(X, V) :-
 (\+ (X = V) -> X #\= V
 ; true
).
```

- Shaving all values in  $X$ 's domain  $\{v_1, \dots, v_n\}$  is the same as performing a constructive disjunction for  $(X = v_1) \vee \dots \vee (X = v_n)$  w.r.t.  $X$

```
shave_values0(X) :-
 fd_set(X, FD), fdset_to_list(FD, L),
 maplist(shave_value(X), L).
% i.e., if L = [A,B,...] this is equivalent to:
% shave_value(X, A), shave_value(X, B), ...
```

- A (slightly more efficient) variant using `findall`:

```
shave(X) :- fd_set(X, FD),
 findall(V, (fdset_member(V,FD), X=V), Vs),
 list_to_fdset(Vs, FD1), X in_set FD1.
```

## An example for shaving, from a kakuro puzzle

- Recall the kakuro puzzle: like a crossword, but with distinct digits 1–9 instead of letters; sums of digits are given as clues.

`% L is a list of N distinct digits 1..9 with sum Sum.`

```
kakuro(N, L, Sum) :-
```

```
 length(L, N), domain(L, 1, 9), all_distinct(L), sum(L,#,Sum).
```

- Example: a 4 letter “word”  $[A,B,C,D]$ , the sum is 23, domains:

```
sample_domains(L) :- L = [A,_,C,D], A in {5,9}, C in {6,8,9}, D=4.
```

```
| ?- L=[A,B,C,D], kakuro(4, L, 23), sample_domains(L).
```

```
 => A in {5}\/{9}, B in (1..3)\/(5..8), C in {6}\/(8..9) ?
```

- Only variable  $B$  gets pruned:

- value 4 is removed by `all_distinct`
- value 9 is removed by `sum`

## An example for shaving, from a kakuro puzzle

- Recall from prev. slide:

```
sample_domains(L) :- L = [A,_,C,D], A in {5,9}, C in {6,8,9}, D=4.
```

```
| ?- L=[A,B,C,D], kakuro(4, L, 23), sample_domains(L).
=> A in{5}\{9}, B in(1..3)\(5..8), C in{6}\(8..9) ?
```

- Shaving 9 off c shows that the value 9 for c is infeasible:

```
| ?- L=[A,B,C,D], kakuro(4, L, 23), sample_domains(L), shave_value(9,C).
=> A in{5}\{9}, B in(2..3)\(5..8), C in{6}\{8} ?
```

- Shaving the whole domain of B leaves just three values:

```
| ?- L=[A,B,C,D], kakuro(4, L, 23), sample_domains(L), shave(B).
=> A in{5}\{9}, B in{2}\{6}\{8}, C in{6}\(8..9) ?
```

- These two shaving operations happen to achieve domain consistency:

```
| ?- kakuro(4, L, 23), sample_domains(L), labeling([], L).
=> L = [5,6,8,4] ? ;
 L = [5,8,6,4] ? ;
 L = [9,2,8,4] ? ; no
```

## When to perform shaving?

- It's often enough to **do it just once, before labeling**
- Recall that labeling is performed for each variable, in a loop
- It may be useful to do shaving in each such loop cycle
  - do your own loop, e.g. use `indomain/1` instead of `labeling/2`
  - use the `value(Goal)` labeling option (not discussed in this course)
- To make shaving efficient one may consider
  - shaving a single variable repeatedly, until a fixpoint is reached (may not pay off)
  - limit it to variables with small enough domain (e.g. of size 2)
  - perform it only after every  $n^{\text{th}}$  labeling step

## Contents

### 3 Declarative Programming with Constraints

- Motivation
- CLPFD basics
- How does CLPFD work
- FDBG
- Reified constraints
- Global constraints
- Labeling
- From plain Prolog to constraints
- Improving efficiency
- Internal details of CLPFD
- Disjunctions in CLPFD
- Modeling**
- User-defined constraints (ADVANCED)
- Some further global constraints (ADVANCED)
- Closing remarks

## Example: the domino puzzle

- See e.g. <http://www.puzzle-dominosa.com/>  
<https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/dominosa...>
- Rectangle of size  $(n + 1) \times (n + 2)$
- A full set of  $n$ -dominoes: tiles marked with  $\{\langle i, j \rangle \mid 0 \leq i \leq j \leq n\}$
- By using each domino exactly once, the rectangle can be covered with no overlaps and no holes
- Input: a rectangle filled with integers  $0..n$  (domino boundaries removed)
- Task: reconstruct the domino boundaries

```
% A puzzle (n=3): % The (only) solution:
1 3 0 1 2 | 1 | 3 0 | 1 | 2 |
3 2 0 1 3 | 3 | 2 0 | 1 | 3 |
3 3 0 0 1 | 3 3 | 0 0 | 1 |
2 2 1 2 0 | 2 2 | 1 2 | 0 |
```

## Modeling – selecting the variables

- Option 1: A matrix of solution variables, each having a value which encodes  $n$ ,  $w$ ,  $s$ ,  $e$ 
  - non-trivial to ensure that each domino is used exactly once
- Option 2: For each domino in the set have variable(s) pointing to its place on the board
  - difficult to describe the non-overlap constraint
- Option 3: Use both sets of variables, with constraints linking them
  - high number of variables and constraints add considerable overhead
- Option 4: Map each gap between – horizontally or vertically – adjacent numbers to a 0/1 variable, where 1 means the mid-line of a domino
  - this is the chosen solution

## Modeling – constraints for option 4

- Let  $S_{yx}$  and  $E_{yx}$  be the variables for the southern and eastern boundaries of the matrix element in row  $y$ , column  $x$ .
- Non-overlap constraint: the four boundaries of a matrix element sum up to 1. E.g. for the element in row 2, column 4 (see blue diamonds below):  
 $\text{sum}([S_{14}, E_{23}, S_{24}, E_{24}], \# =, 1)$
- All dominoes used exactly once: of all the possible placements of each domino, exactly one is used. E.g. for domino  $\langle 0, 2 \rangle$  (see red asterisks below):  
 $\text{sum}([E_{22}, S_{34}, E_{44}], \# =, 1)$

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 3 | 0 | 1 | 2 |
|   |   |   | ◇ |   |
| 3 | 2 | * | 0 | ◇ |
|   |   |   | ◇ | 3 |
| 3 | 3 | 0 | 0 | 1 |
|   |   |   | * |   |
| 2 | 2 | 1 | 2 | * |

## Example for option 4

Case of  $n = 1$ :

|     |     |     |     |   |
|-----|-----|-----|-----|---|
| 1   | E11 | 1   | E12 | 0 |
| S11 | S12 | S13 |     |   |
| 0   | E21 | 0   | E22 | 1 |

% Non-overlap constraint

```

E11 + S11 # = 1 % 1st row
E12 + S12 + E11 # = 1
S13 + E12 # = 1
S11 + E21 # = 1 % 2nd row
S12 + E22 + E21 # = 1
S13 + E22 # = 1

```

% Domino occurrence constraint

```

S11 + S12 + S13 + E12 + E22 # = 1 % 0-1 pairs
E11 # = 1 % 0-0 pairs
E21 # = 1 % 1-1 pairs

```

## Contents

- Declarative Programming with Constraints
  - Motivation
  - CLPFD basics
  - How does CLPFD work
  - FDBG
  - Reified constraints
  - Global constraints
  - Labeling
  - From plain Prolog to constraints
  - Improving efficiency
  - Internal details of CLPFD
  - Disjunctions in CLPFD
  - Modeling
  - User-defined constraints (ADVANCED)**
    - Some further global constraints (ADVANCED)
    - Closing remarks

## User-defined constraints (ADVANCED)

- What should be specified when defining a new constraint:
  - Activation conditions: when should it wake up
  - Pruning: how should it prune the domains of its variables
  - Termination conditions: when should it exit
- Additional issues for reifiable constraints:
  - How should its negation be posted?
  - How to determine whether it is entailed by the store?
  - How to determine whether its negation is entailed by the store?

## Two possibilities for defining new constraints (ADVANCED)

|                                                        | FD predicates                                              | Global constraints                          |
|--------------------------------------------------------|------------------------------------------------------------|---------------------------------------------|
| Number of arguments                                    | Fixed                                                      | Arbitrary (lists of variables as arguments) |
| Specification of pruning logic                         | Using <i>indexicals</i> , a set-valued functional language | In Prolog                                   |
| Specification of activation and termination conditions | Deduced automatically from the indexicals                  | In Prolog                                   |
| Support for reification                                | Yes, using further indexicals                              | No                                          |

## FD predicates – a simple example (ADVANCED)

An FD predicate `'x=<y'` ( $X, Y$ ), implementing the constraint  $x \#=< Y$

- FD clause with neck “+” – pruning rules for the constraint itself:
 

```
'x=<y' (X,Y) +:
 X in inf..max(Y), % intersect X with inf..max(Y)
 Y in min(X)..sup. % intersect Y with min(X)..sup
```
- FD clause with neck “-” – pruning rules for the *negated* constraint:
 

```
'x=<y' (X,Y) -:
 X in (min(Y)+1)..sup,
 Y in inf..(max(X)-1).
```
- FD clause with neck “+?” – the entailment condition:
 

```
'x=<y' (X,Y) +?
 X in inf..min(Y). % X=<Y is entailed if the domain of X
 % becomes a subset of inf..min(Y)
```
- FD clause with neck “-?” – the entailment condition for the negation:
 

```
'x=<y' (X,Y) -?
 X in (max(Y)+1)..sup. % Negation X > Y is entailed when X's
 % domain is a subset of (max(Y)+1)..sup
```

## Defining global constraints (ADVANCED)

The constraint is written as two pieces of Prolog code:

- 1 The start-up code
  - an ordinary predicate with arbitrary arguments
  - should call `fd_global/3` to set up the constraint
- 2 The wake-up code
  - written as a clause of the hook predicate `dispatch_global/4`
  - called by SICStus at activation
  - should return the domain prunings
  - should decide the outcome:
    - constraint exits with success
    - constraint exits with failure
    - constraint goes back to sleep (the default)

## Global constraints – a simple example (ADVANCED)

Defining the constraint  $x \#=< Y$  as a global constraint

## 1 The start-up code

```
lseq(X, Y) :-
 fd_global(lseq(X,Y), void, [min(X),max(Y)]).
%
% constraint name ~~~~ wake-up conditions
%
% initial state
```

## 2 The wake-up code

```
:- multifile clpfd:dispatch_global/4.
:- discontinuous clpfd:dispatch_global/4.
clpfd:dispatch_global(lseq(X,Y), St, St, Actions) :-
 fd_min(X, MinX), fd_max(X, MaxX), % get min of X in MinX, etc.
 fd_min(Y, MinY), fd_max(Y, MaxY),
 (number(MaxX), number(MinY), MaxX =< MinY
 -> Actions = [exit]
 ; Actions = [X in inf..MaxY, Y in MinX..sup]
).
```

The start-up predicate `fd_global/3` (ADVANCED)

- `fd_global(Constraint, State, Susp)`: start up constraint `Constraint` with initial state `State` and wake-up conditions `Susp`.
  - `Constraint` is normally the same as the head of the start-up predicate
  - `State` can be an arbitrary non-variable term
  - `Susp` is a list of terms of the form:
    - `dom(X)` – wake up at any change of domain of variable `x`
    - `min(X)` – wake up when the lower bound of `x` changes
    - `max(X)` – wake up when the upper bound of `x` changes
    - `minmax(X)` – wake up when the lower or upper bound of `x` changes
    - `val(X)` – wake up when `x` is instantiated

The wake-up hook predicate `dispatch_global/4` (ADVANCED)

- `dispatch_global(Constraint, State0, State, Actions)`: When `Constraint` is woken up at state `State0` it goes to state `State` and executes `Actions`
  - `Actions` is a list of terms of the form:
    - `exit` – the constraint will exit with success
    - `fail` – the constraint will exit with failure
    - `X=V, X in R, X in_set S` – the given pruning will be performed
    - `call(Module:Goal)` – the given goal will be executed
- No pruning should be done inside `dispatch_global`, instead the pruning requests should be returned in `Actions`
- States can be used to share information between invocations of the constraint
- Information about the domain variables can be queried using reflection predicates

## Contents

- 3 Declarative Programming with Constraints
  - Motivation
  - CLPFD basics
  - How does CLPFD work
  - FDBG
  - Reified constraints
  - Global constraints
  - Labeling
  - From plain Prolog to constraints
  - Improving efficiency
  - Internal details of CLPFD
  - Disjunctions in CLPFD
  - Modeling
  - User-defined constraints (ADVANCED)
  - Some further global constraints (ADVANCED)
  - Closing remarks

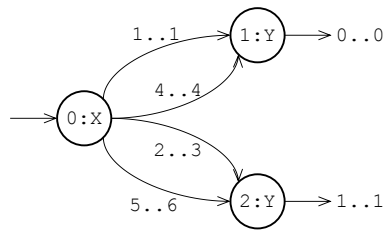
## Specifying a relation using a DAG (ADVANCED)

- `case(Template, Tuples, Dag[, Options])`: uses a directed acyclic graph (DAG), the nodes of which correspond to variables in the same order as they appear in `Template` and arcs are labeled with admissible intervals of the variable of the arc's starting node. For each tuple in `Tuples`, there must be an appropriate path from the root node to a leaf node.

Example: A is in [1,6], B is in [0,1]; if dividing A by 3 gives remainder 1, then B is even, otherwise B is odd.

```
?- case([X,Y],[[A,B]],[node(0,X,[(1..1)-1,(2..3)-2,(4..4)-1,(5..6)-2]),
 node(1,Y,[0..0]),node(2,Y,[1..1])]),
 labeling([], [A,B]),write(A-B),write(' '),fail.
```

⇒ 1-0 2-1 3-1 4-0 5-1 6-1



## Specifying a relation using an automaton (ADVANCED)

- `automaton(Signature, SourcesSinks, Arcs)`: `SourcesSinks` and `Arcs` define a finite automaton that classifies ground instances as solutions or non-solutions. The constraint holds if the automaton accepts the list `Signature`.

Example: the first few elements (at least one) of `L` must be all 1, the remaining elements (at least one) must be all 2.

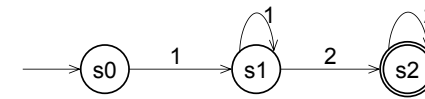
```
| ?- length(L,4), automaton(L,[source(s0),sink(s2)],
 [arc(s0,1,s1),arc(s1,1,s1),arc(s1,2,s2),arc(s2,2,s2)]),
 labeling([],L).
```

L = [1,1,1,2] ? ;

L = [1,1,2,2] ? ;

L = [1,2,2,2] ? ;

no



## Contents

### 3 Declarative Programming with Constraints

- Motivation
- CLPFD basics
- How does CLPFD work
- FDBG
- Reified constraints
- Global constraints
- Labeling
- From plain Prolog to constraints
- Improving efficiency
- Internal details of CLPFD
- Disjunctions in CLPFD
- Modeling
- User-defined constraints (ADVANCED)
- Some further global constraints (ADVANCED)
- Closing remarks

## What else is there in SICStus Prolog?

- Further constraint libraries:
  - CLPB – booleans
  - CLPQ/CLPR – linear inequalities on rationals/reals
  - Constraint Handling Rules: generic constraints
- Other features
  - “Traditional” built-in predicates, e.g. sorting, input/output, exception handling, etc.
  - Powerful data structures, e.g. AVL trees, multisets, heaps, graphs, etc.
  - Definite clause grammars, an extension of context-free grammars with Prolog terms
  - Interfaces to other programming languages, e.g. C/C++, Java, .NET, Tcl/Tk
  - Integrated development environment based on Eclipse (Spider)
  - Execution profiling
  - ...

## Some applications of (constraint) logic programming

- Boeing Corp.: Connector Assembly Specifications Expert (CASEy) – an expert system that guides shop floor personnel in the correct usage of electrical process specifications.
- Windows NT: `\WINNT\SYSTEM32\NETCFG.DLL` contains a small Prolog interpreter handling the rules for network configuration.
- Experian (one of the largest credit rating companies): Prolog for checking credit scores. Experian bought Prologia, the Marseille Prolog company.
- IBM bought ILOG, the developer of many constraint algorithms (e.g. that in `all_distinct`); ILOG develops a constraint programming / optimization framework embedded in C++.
- IBM uses Prolog in the Watson deep Question-Answer system for parsing and matching English text