Part II

Declarative Programming with Constraints

Contents

Block declarations

- A block declaration causes a goal to suspend when certain arguments are unbound
- Multiple block declarations mean disjunctive conditions
- Examples:

  `:- block p(-, ?, -, ?).` suspends when 1st and 3rd arg is unbound.

  `:- block p(-, ?), p(?, -).` suspends when 1st or 2nd arg is unbound.
- When blocking condition(s) hold no more, the call executes immediately.
- Example: safe append, with no infinite choice points:

```
:- block app(-, ?, -).
% app(L1, L2, L3):  The concatenation of L1 and L2 is L3.
% Blocks if *both* L1 and L3 are unbound variables.
app([], L, L).
app([X|L1], L2, [X|L3]) :-      app(L1, L2, L3).

| ?- app(L1, L2, L3).              ⟹    user:app(L1,L2,L3) ? ; no
| ?- app(L1, L2, L3), L3 = [1|_].
          ⟹    L1 = [], L2 = [1|_A], L3 = [1|_A] ? ;
          ⟹    L1 = [1|_A], L3 = [1|_B], user:app(_A,L2,_B) ? ; no
```

Further coroutining predicates

- `freeze(X, Goal)`: `Goal` is true. Suspends `Goal` until `X` is instantiated.
- `dif(X, Y)`: `X` and `Y` are different. Suspends until this can be decided.
- `when(Condition, Goal)`: `Goal` is true. Suspends `Goal` until `Condition` is satisfied. Here `Condition` is a very simple Prolog goal:

```
Condition ::=  nonvar(X) | ground(X) | ?=(X,Y) |
               Condition, Condition |
               Condition; Condition
```

  - For example, `process/3` will execute only when `T` is ground and either `X` is not a variable, or the unifiability of `X` and `Y` can be decided:

```
| ?- when( (ground(T),(nonvar(X);?=(X,Y))),
           process(X,Y,T)).
```

- `frozen(X, Goals)`: `Goals` are the goals suspended because of variable `X`.
- `call_residue_vars(Goal, Vars)`: `Vars` is the list of variables because of which goals were suspended during the execution of `Goal`.

The N-queens problem

- Place N queens on a NxN chessboard so that no two queens attack each other
- The Prolog list `[`Q_1`, ..., `Q_N`]` represents the placement: row *i* contains a queen in column Q_i.
- The "generate and test" approach for solving N-queens

```
:- use_module(library(between)).
:- use_module(library(lists),[select/3]).

queens_gt(N, Qs):-
    findall(I, between(1, N, I), L),    % L = [1,2,...N]
    permutation(L, Qs),
    safe(Qs).   % Placement Qs has no queens attacking each other

% permutation(L, P): P is a permutation of L.
permutation([], P) :- !, P = [].
permutation(L0, [X|P]) :-
    select(X, L0, L1),  % select the 1st elem of permutation
    permutation(L1, P). % permute the rest
```

Safe placement of N-queens

- The code to check if a placement is safe

```
% safe(Qs): Placement Q is safe from queens attacking each other
safe([]).
safe([Q|Qs]):-
    no_attack(Qs, Q, 1), safe(Qs).

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

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

Using coroutining for fusing generate and test

- Add blocking conditions to the test for `no_threat`:

```
:- block no_threat_b(-,?,?), no_threat_b(?,-,?).
% Queens placed in column X of row n, and in column Y of row n+I
% do not attack each other. Wakes up when all args are instantiated.
no_threat_b(X, Y, I) :-
    /* X =\= Y, */ Y =\= X-I, Y =\= X+I.
```

- `no_attack_b` is the same as `no_attack`, but using `no_threat_b`, `safe_b` is the same as `safe`, but using `no_attack_b`
- "First" test, then generate:

```
queens_tg(N, Qs):-
    length(Qs, N),        Qs is a list of N vars
    safe_b(Qs),           set up blocked no_threat checks
    findall(I, between(1, N, I), L),
    permutation(L, Qs). When Qs_i is bound, it is checked
                        immediately against queens placed earlier
```

- The ratio of runtimes of `gt` and `tg` variants
  `N` →ratio: 8 → 4, 9 → 8, 10 → 16, 11 → 32, 12 → 69, 13 → 154

Contents

About CLP in general

- The CLP($\mathcal{X}$) schema

Prolog or some other prog. language, e.g. C++	+	"strong" reasoning capabilities on a restricted domain $\mathcal{X}$ involving specific constraint (relation) and function symbols.

- Examples for the domain $\mathcal{X}$:
  - $\mathcal{X} =$ Q or R (rationals or reals) – cf. SICStus libraries `clpq`, `clpr`
    constraints: linear equalities and inequalities
    reasoning techniques: Gauß elimination and the simplex method
  - $\mathcal{X} =$ FD (Finite Domains, e.g. of integers) – `library(clpfd)`
    constraints: various arithmetic, logic, and combinatorial relationships
    reasoning techniques: methods developed for solving Constraint Satisfaction Problems (CSPs), a branch of Artificial Intelligence (AI)
  - $\mathcal{X} =$ B (Boole values `true` and `false`, or `1` and `0`) – `library(clpb)`
    constraints: relations of propositional calculus (negation, conj., etc.)
    reasoning techniques: AI methods developed for solving SAT (propositional satifiability) problems, e.g. binary Decision Diagrams.

Constraint Satisfaction Problems

- A CSP task is a triple (X, D, C)
  - $X = \langle x_1, \ldots, x_n \rangle$ – variables
  - $D = \langle D_1, \ldots, D_n \rangle$ – domains, i.e. nonempty finite sets
  - variable x_i can take its values from the set D_i, called the domain of x_i
  - C is the set of constraints (primitive relations), with arguments from X (e.g. $C \ni c = r(x_1, x_3)$, $r \subseteq D_1 \times D_3$)
- The solution of a CSP task: the assignment, to each x_i, of a value $v_i \in D_i$ in such a way that all $c \in C$ constraints are simultaneously satisfied.
- **Definition:** A value $d_i \in D_i$ of a variable x_i is infeasible w.r.t. the constraint $c = r(\ldots, x_i, \ldots)$, if no assignment can be found for the remaining variables of c, which, taken with d_i, satisfies c.
- **Proposition:** removing an infeasible value (domain pruning) does not affect the set of solutions.
- **Definition:** A constraint is *arc-consistent*, if no variable domain contains an infeasible value. A CSP task is *arc-consistent*, if all constraints in it are arc-consistent. (A task with binary constraints only can been seen as a graph: var $\Rightarrow$ node, relation $\Rightarrow$ arc – hence the name arc-consistency.)
- Arc-consistency can be ensured by repeated domain pruning.

The process of solving CSP (CLPFD) problems

- Modelling – 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.
- Optimisation, e.g. redundant constraints, labeling heuristics, constructive disjunction, shaving.
- Generate-and-test (G&T) vs. the CSP (constrain-and-generate) approach (4-queens example)
  - G&T:
    ```
    ?- between(1,4,A),between(1,4,B),between(1,4,C),...
       A=\=B,A=\=B+1,A=\=B-1,A=\=C+2,...
    ```
  - CSP:
    ```
    ?- L=[A,B,C,D],domain(L,1,4),
       A#\=B,A#\=B+1,A#\=B-1,A#\=C+2,...,labeling([ff],L).
    ```

Contents

`library(clpfd)` – an overview

- Domain: a finite set of integers (allowing the restricted use of infinite intervalls 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, e.g. indexicals and global constraints

Some important constraints

- Membership constraints
  - `domain(+Vars, +Min, +Max)` where
    `Min`: ⟨ `integer` ⟩ or `inf` ($-\infty$),
    `Max`: ⟨ `integer` ⟩ or `sup` ($+\infty$):
    All elements of list `Vars` belong to the closed 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	::=	*ConstantSet*	
		| *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}`.

Some important constraints

- Arithmetic formula constraints: *Expr* `Relop` *Expr* where

```
Expr ::=  ⟨integer⟩ | ⟨variable⟩
       | - Expr | Expr + Expr | Expr - Expr | Expr * Expr
       | Expr / Expr                        % integer division
       | Expr mod Expr  | Expr rem Expr     % differ only for ints < 0
       | min(Expr,Expr) | max(Expr,Expr) | abs(Expr)
RelOp ::= #= | #\= | #< | #=< | #> | #>=
```

- Global arithmetic constraints (global = having arbitrary number of args):
  - `sum(+Xs, +RelOp, ?Value)`: Σ `Xs` *Relop* `Value`.
  - `scalar_product(+Coeffs, +Xs, +RelOp, ?Value[, +Options])` (last arg. optional): Σ `(Coeffs*Xs)` *RelOp* `Value`.
  - `minimum(?V, +Xs)`, `maximum(?V, +Xs)`: `V` is the min/max of `Xs`.
- Some combinatorial (global) constraints:
  - `all_different([`X_1`,...,`X_n`])`: same as X_i `#\=` X_j for all $0 < i < j \leq n$.
  - `all_distinct([`X_1`,...,`X_n`])`: same as `all_different`, but guarantees arc-consistency between the *n* variables.

```
| ?- 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
```

Labeling

- Labeling: search by systematic assignment of feasible values to vars.
- `indomain(?Var)`: `Var` is assigned feasible values in ascending order.
- `labeling(+Options, +Vars)`: assigns all `Vars`. Some useful options:
  - Variable selection: Select for assignment ...
    - `leftmost` (default) – the leftmost unbound variable;
    - `ff` (first-fail) – the leftmost variable with the smallest domain;
    - `ffc` – the leftmost variable with the smallest domain which participates in most constraints;
    - `min`/`max` – the leftmost variable with the smallest lower bound/largest upper bound;
    - `variable(`*Sel*`)` – the var. prescribed by the user through *Sel*.
  - Direction: `up`(default)/`down` – assign in ascending/descending order.
  - Value selection: Create a choice-point distinguishing between ...
    - `step` (default) – the lower/upper bound and the rest (2-way);
    - `enum` – all values in the domain (*n*-way, *n* is the domain size);
    - `bisect` (domain spliting) – the two halves of the domain (2-way);
    - `value(`*Enum*`)` – as prescribed by the user through *Enum*.

The implementation of CLPFD

- The main data structure: the backtrackable constraint store – maps variables to their domains.
- Simple constraints: e.g. `X in 1..10` or `X #< 10` just modify the store.
- Composite constraints are implemented as daemons, which keep removing infeasible values from argument domains, e.g., given the store `X in 1..6, Y in {1,6,7,8,9}` the daemon for `X+5 #= Y` removes $\Longrightarrow$ `5`, `6` from `X`; `1` from `Y`, producing: `X in 1..4, Y in 6..9`. (*)
- Composite constraints can provide reasoning of different strengths:
  - domain-consistency ($\equiv$ arc-cons.) – removes all infeasible values, e.g. after (*), `Y #\= 7` removes $\Longrightarrow$ `2` from `X`. (**)
    (Cost: exponential in the number of variables.)
  - bound-consistency – (repeatedly) removes infeasible bounds only, i.e. *middle* elements, as in (**), are not removed. E.g. after (*), `Y #< 8` $\Longrightarrow$ remove `4` and then `3`, the upper bounds of `X`, so that all bounds become feasible. (Cost: linear in the # of vars)
- A daemon may exit (die), when the constraint it represents is entailed by (follows from) the constraint store; e.g. `X #< Y` may exit if the store contains: `X in 1..5 and Y in 7..9`.

The implementation of CLPFD (contd.)

- To execute a constraint *C*:
  - Transform *C* (at compile time) to a series of primitive constraints, e.g. `X*X #> Y` ⇒ `X*X #= Z, Z #> Y` (formula constraints only).
  - Post *C* (or each of the primitive constraints obtained from *C*):
    - execute immediately (e.g. `X #< 3`); or
    - create a daemon for *C*:

      tell the constraint engine when to wake me up (activation conditions)
      prune the domains
      until the termination condition becomes true **do**
      go to sleep (wait for activation)
      prune the domains
      enduntil
- An activation condition: the domain of a variable `X` changes in SOME way: SOME = any; lower/upper bound change; instantiation; . . .
- The termination condition is constraint specific; earliest: when the constraint is entailed; latest: when all its variables are instantiated.
- `| ?- assert(clpfd:full_answer).`
  makes SICStus Prolog show the non-terminated constraints.

The implementation of some constraints

- `A #\= B` (arc-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.
- `A #< B` (arc-consistent)
  - **Activation**: when `min(A)` (the lower bound of `A`) or `max(B)` (the upper bound of `B`) changes.
  - **Pruning**:
    remove from the domain of `A` all x's such that $x \geq$ `max(B)`,
    remove from the domain of `B` all y's such that $y \leq$ `min(A)`.
  - **Termination**: if one of the variables `A` and `B` becomes instantiated (could be improved).

The implementation of some constraints (contd.)

- `X+Y #< T` (bound-consistent)
  - **Activation**: when the lower or upper bound changes for any of the variables `X`, `Y`, `T`.
  - **Pruning**:
    narrow the domain of `T` to `(min(X)+min(Y))..(max(X)+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.
- `all_distinct([`A_1`,...])` (arc-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 paths in bipartite graphs).
  - **Termination**: when at most one of the variables is uninstantiated.

Consistency levels guaranteed by SICStus Prolog

- Membership constraints (trivially) ensure arc-consistency.
- Linear arithmetic constraints ensure at least bound-consistency.
- Nonlinear arith. 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}, Z #= X-2.                 ⟹  Z in{2}\/{7} ?
                                           ⟹  Domain consistent
| ?- X in {4,9}, Y in {2,3},
     scalar_product([1,-1], [X,Y], #=, Z, [consistency(domain)]).
                                           ⟹  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
```

Contents

Reified constraints – an introduction

- A speculative solution for counting the positive members of a list:

```
% pcount0(L, N): L has N positive elements.
pcount0([X|L], N) :- (   X #> 0, N1 #= N-1, pcount0(L, N1)
                     ;   X #=< 0, pcount0(L, N)
                     ).
pcount0([], 0).
```

```
| ?- pcount0([A,B], 0).  ⟹  A in inf..0, B in inf..0 ? ; no
| ?- pcount0([A,B], 1).  ⟹  A in 1..sup, B in inf..0 ? ;
                         ⟹  B in 1..sup, A in inf..0 ? ; no
```

- Speculative disjunction: ChPs made while posting constraints.

  `| ?- length(L, 20), pcount0(L, 0).`   takes ∼ 15 sec!

- With reification a non-speculative solution is possible (no ChPs):

```
% pcount1(L, N): L has N positive elements.
pcount1([X|Xs], N) :-
        (X #> 0) #<=> B,          % (X > 0) iff (B = 1)
        N1 #= N-B, pcount1(Xs, N1).
pcount1([], 0).
```

Reification – obtaining the truth value of a constraint

- The reification of an n-ary constraint $C = r(x_1, \ldots, x_n)$
  - is a constraint $r_{reif}(x_1, \ldots, x_n, b)$ defined as $r(x_1, \ldots, x_n) \Leftrightarrow b = 1$, where b is a Boolean (0-1) variable;
  - denoted by C `#<=> B`
  - this means that `B in 0..1` and `B` is `1` iff C is true.
- A reified constraint is very much like an arithmetic constraint, and sometimes can be replaced by an arithmetic one, e.g.
  - `(X #> 0) #<=> B` means: `B` is the truth value of `X > 0`.
  - If `X` has the domain `0..9`, then `B #= (X+9)/10` is the very same constraint, just implemented differently
- Which constraints can be reified?
  - Arithmetic formula constraints (`#=`, `#=<`, etc.)
  - Membership constraints (`in`)
- Global constraints (e.g. `all_distinct/1, sum/3`) cannot be reified.

Executing reified constraints

- The execution of C `#<=> B` requires the following actions:
  - When `B` is instantiated:
    - if `B=1`, post C, if `B=0`, post $\neg C$,
  - When C is entailed (i.e. the store implies C), set `B` to `1`.
  - When C is disentailed (i.e. $\neg C$ is entailed), set `B` to `0`.
- The above three action types are delegated to three daemons.
- Detecting entailment can be done with different levels of precision:
  - A reified membership constraint C detects domain-entailment, i.e. `B` is set as soon as C is a consequence of the store
  - A linear arithmetic constraint C is guaranteed to detect bound-entailment, i.e. `B` is set as soon as C is a consequence of the interval closure of the store. (The interval closure is obtained by removing the holes in the domains.) E.g. the store below implies the constraint $\mathtt{X}+\mathtt{Y}\neq\mathtt{Z}$ (odd+even$\neq$even), but its interval closure does not:

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

  - When a constraint becomes ground, its (dis)entailment is detected

Propositional constraints

- Propositional connectives allowed by SICStus Prolog CLPFD:

`#\ Q`	negation	`op(710, fy, #\ ).`
`P #/\ Q`	conjunction	`op(720, yfx, #/\ ).`
`P #\ Q`	exclusive or	`op(730, yfx, #\ ).`
`P #\/ Q`	disjunction	`op(740, yfx, #\/ ).`
`P #=> Q`	implication	`op(750, xfy, #=> ).`
`Q #<= P`	implication	`op(750, yfx, #<= ).`
`P #<=> Q`	equivalence	`op(760, yfx, #<=>).`

- 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.
- The propositional constraints are
  - built from vars, ints and reifiable constraints using above operators;
  - executed by transforming them to arith. and reified constraints, e.g.

    `(X#>0) #\/ (Y#<5).` $\Longleftrightarrow$ `(X#>0)#<=>B1, (Y#<5)#<=>B2, B1+B2#>0.`

N-queens – a CLPFD variant

- Recall: the Prolog list `[Q1, ..., QN]` represents the placement: row i contains a queen in column Q_i.

```
% queens(+Lab, +N, ?Q): Q is a safe placement i.e. no two queens
% attack each other, obtained using the labeling options Lab
queens(Lab, N, Qs) :-
    length(Qs, N), domain(Qs, 1, N),
    safe(Qs), labeling(Lab, Qs).

% safe(Qs): List Qs describes a safe placement of queens.
safe([]).
safe([Q|Qs]):- no_attack(Qs, Q, 1), safe(Qs).

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

% Queens in row n, col X and in row n+I, col Y do not attack each other.
no_threat(X, Y, I) :- Y #\= X, Y #\= X-I, Y #\= X+I.
```

Contents

User-defined constraints

- 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 decide its entailment?
  - How to decide its disentailment (the entailment of its negation)?

Possibilities for defining constraints

- Global constraints
  - Arbitrary number of arguments (variable lists as arguments)
  - Activation, pruning, termination is specified by arbitrary Prolog code.
  - No specific support for reification
- FD predicates
  - Fixed number of arguments
  - Support for reification
  - Pruning and entailment is specified by so called indexicals (Pascal van Hentenryck), using a set-valued functional language.
  - Activation and termination conditions deduced automatically from the indexicals.

FD predicates – a simple example

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

- FD clause with neck `+:` – the prunings 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 `-:`– the prunings 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=<Y is entailed if the domain of X
    X in inf..min(Y).       % becomes a subset of  inf..min(Y)
```

- FD clause with neck `-?` – the entailment condition for the negation:

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

Defining global constraints

- The constraint is written as two pieces of Prolog code
  - The start-up code
    - an ordinary predicate with arbitrary arguments
    - should call `fd_global/3` to set up the constraint
  - The wake-up code
    - written as a clause (of a hook predicate) to be called by SICStus at activation
    - should return the domain prunings
    - should decide whether the constraint can exit (with success or failure) or should fall asleep and keep pruning later

Global constraints – a simple example

Defining the constraint `X #=< Y` as a global constraint

- The start-up code

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

- The wake-up code

```
:- multifile clpfd:dispatch_global/4.
:- discontiguous clpfd:dispatch_global/4.
clpfd:dispatch_global(lseq(X,Y), St, St, Actions) :-
        dispatch_lseq(X, Y, Actions).

dispatch_lseq(X, Y, 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]
    ).
```

Contents

Shaving – a special case of constructive disjunction

- Base idea: remove v from X if setting $X = v$ fails immediately, i.e. without labeling
- Shaving a single value `V` off the domain of `X` is similar to a constructive disjunction $(X = v) \lor (X \neq v)$ w.r.t. X

```
shave_value(V, X) :-
        \+ X = V, !, X in \{V}.
shave_value(_, _).
```

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

```
shave_all(X) :-
         fd_set(X, FD), fdset_to_list(FD, L),
         findall(X, member(X,L), Vs),
         list_to_fdset(Vs, FD1), X in_set FD1.
```

- Shaving may be applied repeatedly, until a fixpoint (may not pay off)
- Shaving is normally done during labeling. To reduce its costs, one may:
  - limit it to variables with small enough domain (e.g. of size 2)
  - perform it only after each n^{th} labelling step (requires global variables)

An example for shaving, from a kakuro puzzle

- 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 a 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.
```

- Only `B` gets pruned: 4, because of `all_distinct`, 9 because of `sum`

```
| ?- 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 the value 9 for `C` is infeasible:

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

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

```
| ?- L=[A,B,C,D], kakuro(4, L, 23), sample_domains(L), shave_all(B).
     ⟹  ..., B in{2}\/{6}\/{8}, ... ?
```

- These two shaving operations happen to achieve domain constistency:

```
| ?- 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
```

Contents

The domino puzzle

- Consider a rectangle of size $(n+1) \times (n+2)$ covered by a full set of n-dominoes with no overlaps and no holes.
- The set has tiles of size 1×2, marked with $\{\langle i, j \rangle | 0 \leq i \leq j \leq n\}$
- Input: a rectangle filled with integers $0..n$ (domino boundaries removed)
- The task is to reconstruct the domino boundaries[13]

```
%  A puzzle (n=3):    % The (only) solution:  % The solution matrix:
                      ---------------------
1   3   0   1   2     | 1 | 3   0 | 1 | 2 |       n   w   e   n   n
                      |   |-------|   |   |
3   2 . 0   1   3     | 3 | 2   0 | 1 | 3 |       s   w   e   s   s
                      |---------------|---|
3   3   0   0   1     | 3   3 | 0   0 | 1 |       w   e   w   e   n
                .     |-------|-------|   |
2   2   1   2 . 0     | 2   2 | 1   2 | 0 |       w   e   w   e   s
                      ---------------------
```

- The input: a matrix of integers, the solution: a matrix of atoms `n, w, s, e` (the given position is a northern, western, . . . part of a domino piece)

[13] Red dots show the 3 possible placements of the mid-line of domino $\langle 0, 2 \rangle$

Modelling the domino puzzle

Selecting the variables[14]

- Option a: A matrix of solution variables, each having a value which encodes `n, w, s, e` – difficult to ensure that each domino is used once
- Option b: for each domino in the set have variable(s) pointing to its place on the board – makes difficult to describe the non-overlap constraint
- Solution 1: use both sets of variables, with constraints linking them
- Solution 2: Map each gap between (horiz. or vert.) adjacent numbers to a 0-1 variable, whose value is 1, say, iff it is the mid-line of a domino

Selecting the constraints (for Solution 2):

- Let `Syx` and `Eyx` 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: `sum([S14,E23,S24,E24], #=, 1)`
- All dominoes used: Of all the possible placements of each domino exactly one is used. E.g. for the domino $\langle 0, 2 \rangle$ (see red dots on the previous slide): `sum([E22,S34,E44], #=, 1)`

[14]SICStus `library('clpfd/examples/dominoes')` contains a solution using the `geost/2` constraint.

Contents

What else is there in (SICStus) Prolog?

- Prolog features
  - Definite clause grammars,
    e.g. `expr -> term, ( ("+"|"-"), expr | "" ).`
  - “Traditional” built-in predicates,
    e.g. sorting, input/output, exception handling, etc.
- Constraints
  - Further constraint libraries in SICStus:
    - CLPB – booleans
    - CLPQ/CLPR – linear (in/dis)equalities on rationals/reals
    - Constraint Handling Rules: generic constraints,
      this package is also available for other host-languages, e.g. Java.
- Numerous other libraries

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