Constraint Programming
with FaCiLe

Nicolas Barnier
nicolas.barnier@enac.fr

École Nationale de l'Aviation Civile

2018 – 2019

Objectives

- Master the fundamental building blocks of a non-Prolog Constraint Programming library
- Be able to **model a combinatorial problem** as a constraint program and efficiently **develop a solver**
- Experiment with **search strategies**
- Basic knowledge of symmetry breaking and redundant constraints

Plan

Program

FaCiLe: Functional Constraint Library

FaCiLe

- Developed at the MAIAA lab @ ENAC
- Written in OCaml
- Not publicly updated since a long time but still maintained for research projects
- Terse documentation built from the comments in `.mli` interface files
- This course is based on the latest development version which slightly differs from the last available documentation (version 1.1)

Programming with FaCiLe

Interpreter and compilers

- The library can be used with the OCaml interpreter `ocaml` to experiment with the basics or test small pieces of code

  ```
  ocaml -I path facile.cma
  ```

  where `path` points to the installation directory
- Or constraint programs can be compiled with `ocamlc` and `ocamlopt`:

  ```
  ocamlc -o solver -I path facile.cma cp.ml
  ocamlopt -o solver -noassert -I path facile.cmxa cp.ml
  ```
- Option `-noassert` should be used to avoid time-consuming integer overflow checks set by default

Basic Example

Cryptarithmetic

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

- Each letter corresponds to a distinct digit.
- $M \neq 0, S \neq 0$

Encoding

- 8 variables
- 1 equation
- 2 disequations
- 1 “all different” global constraint

My First Program with FaCiLe

```
open Facile open Easy
let solve = fun () ->
  (* variables *)
  let [|s;e;n;d;m;o;r;y|] as vars = Fd.array 8 0 9 in
  (* constraints *)
  Cstr.post (fd2e s <>~ i2e 0);
  Cstr.post (fd2e m <>~ i2e 0);
  Cstr.post (Alldiff.cstr vars);
  let send =
    i2e 1000 *~ fd2e s +~ i2e 100 *~ fd2e e +~ i2e 10 *~ fd2e n
      +~ fd2e d in
  let more =
    i2e 1000 *~ fd2e m +~ i2e 100 *~ fd2e o +~ i2e 10 *~ fd2e r
      +~ fd2e e in
  let money =
    i2e 10000 *~ fd2e m +~ i2e 1000 *~ fd2e o +~ i2e 100 *~ fd2e n
      +~ i2e 10 *~ fd2e e +~ fd2e y in
  Cstr.post (send +~ more =~ money);
```

My First Program with FaCiLe

```
  (* search goal *)
  let goal = Goals.Array.labeling vars in
  (* resolution *)
  if Goals.solve goal then
    let [|s;e;n;d;m;o;r;y|] = Array.map Fd.elt_value vars in
    Printf.printf "   %d%d%d%d\n+  %d%d%d%d\n= %d%d%d%d%d\n"
      s e n d m o r e m o n e y
  else
    Printf.printf "No solution found\n"

let () =
  solve ()
```

```
barnier@venar:~/cours/cours_facile_eng$ ./crypt.opt
   9567
+  1085
= 10652
```


Library Structure

State

The state of FaCiLe (goal stack, constraint queue...) is stored in a global variable and therefore does not need to be passed as a parameter to the functions of the library

Module Facile

The user interface to the library is provided by module `Facile` which gathers several specialized submodules (`Domain`, `Cstr`, `AllDiff`, `Goals`...) and should be opened systematically

Module Easy

Most frequently used functions and operators and submodule Fd (variables operations) can then be directly accessed by opening module Easy

Naming conventions

Type

The main **type** of a module `M` is named `M.t`:
`Fd.t`, `Cstr.t`, `Goals.t`...

Printing

Most modules of FaCiLe have a print function `M.fprint` which takes an out channel as parameter and can be used with the `%a` format directive of module `Printf`:

```
let x = Fd.interval 2 12;;
Fd.fprint stdout x;;
_0[2..12]- : unit = ()
Printf.printf "x=%a\n" Fd.fprint x;;
x=_0[2..12]
- : unit = ()
```


Variables

Module Fd

Variables (unknown) of type `Fd.t` of the constraint program are associated with finite domains of integers. Variables can be created by providing (with type `elt = int`):

- a domain: `create: Domain.t -> t`
- the bounds of an interval: `interval: elt -> elt -> t`
- an array size and interval bounds:
  `array: string -> int -> elt -> elt -> t array`
- an integer (assigned variable): `elt: elt -> t`
- an expression of type `Arith.t`: `Arith.e2fd: t -> Fd.t`

The domain of a variable can be modified by value removal and assignment. These operations are performed by constraints and goals.

Variables

Operations

- Access to the assignment value (raises an exception if the variable is not assigned): `elt_value: t -> elt`
- Assignment checks: `is_var, is_bound: t -> bool`
- Algebraic interface: `value: t -> (domain,elt) concrete`

```
match Fd.value x with
  Val v -> ...
| Unk dom -> ...
```

- Extrema: `min,max: t -> elt`, `min_max: t -> elt * elt`
- Cardinal (with type `size = int`): `size: t -> size`
- Membership: `member: t -> elt -> bool`
- Printing: `fprint: out_channel -> t -> unit`
  `fprint_array: out_channel -> t array -> unit`

Variables

Variable creation and printing

With the OCaml interpreter:

```
# open Facile open Easy;;
# let x = Fd.interval (-42) 1729;;
val x : Facile.Easy.Fd.t = <abstr>
# Printf.printf "x: %a\n" Fd.fprint x;;
x: _0[-42..1729]
- : unit = ()
# let dom = Domain.create [1;2;3;5;7;8;20;21];;
val dom : Facile.Domain.t = <abstr>
# let y = Fd.create dom;;
# Fd.fprint stdout y;;
_1[1..3;5;7..8;20..21]- : unit = ()
# Fd.fprint stdout (Fd.elt 7);;
_2[7]- : unit = ()
```

Variables are printed as sequences of disjoint intervals

Variables

Arrays of variables

Arrays of variables are often needed in constraint programs:

```
# let vars = Fd.array 4 0 9;;
val vars : Facile.Easy.Fd.t array =
  [|<abstr>; <abstr>; <abstr>; <abstr>; <abstr>|]
# Fd.fprint_array stdout vars;;
[|_2[0..9]; _3[0..9]; _4[0..9]; _5[0..9]|]- : unit = ()
# Fd.is_var vars.(0);;
- : bool = true
# Fd.min_max vars.(1);;
- : Facile.Easy.Fd.elt * Facile.Easy.Fd.elt = (0, 9)
# Fd.size vars.(2);;
- : Facile.Easy.Fd.size = 10
# Fd.elt_value vars.(3);;
Exception: Failure "Fatal error: XxxFd.elt_value:
                    unbound variable _5[0-9]".
```


Domains

Module Domain

Finite domains are set-like structures of type `Domain.t` and can be created with:

- the empty domain: `empty: t`
- a list of integers: `create: elt list -> t`
- the bounds of an interval: `interval: elt -> elt -> t`

Many set operations are supported: minimum and maximal values, cardinal, insertion and deletion, union, intersection, difference...

Mainly for advanced usage:

- most constraint programs use variables with interval domains returned by `Fd.interval` or `Fd.array`
- used in the implementation of user-defined (and built-in) constraints

Arithmetic Expression

Module Arith

Arithmetic expressions of type `Arith.t` are build with variables and integer constants through conversion functions and combined with **specific** operators suffixed with '~'

Provided by module Easy

- Conversion functions:
  - integer to expression: `i2e: int -> Arith.t`
  - variable to expression: `fd2e: Fd.t -> Arith.t`
- Operators: `+~ *~ -~ /~ %~ **~`
  Note: there is no unary minus operator (use `i2e 0 -~ expr` instead) and the exponent of the power operator must be an integer constant

```
#i2e 2 *~ fd2e x -~ fd2e y **~ 2 +~ i2e 3;;
- : Facile.Arith.t = <abstr>
```


Arithmetic Expression

Provided by module Arith only (not in module Easy)

- Absolute value: `abs: t -> t`
- Array operations:
  - sum:
    - `sum: t array -> t`
    - `sum_fd: Fd.t array -> t`
  - scalar product:
    - `scalprod: int array -> t array -> t`
    - `scalprod_fd: int array -> Fd.t array -> t`
- Conversion to variable: `e2fd: t -> Fd.t`

```
# let t1 = Array.init 10 (fun i -> i+1);;
# let t2 = Fd.array 10 1 10;;
# let ps = Arith.e2fd (Arith.scalprod_fd t1 t2);;
val ps : Facile.Var.Fd.t = <abstr>
# Printf.printf "ps=%a\n" Fd.fprint ps;;
ps=_24[55..550]
```

Arithmetic Constraints

Provided by module Easy

Constraints of type `Arith.t -> Arith.t -> Cstr.t`:

- equation: `=~`
- disequation: `<>~`
- inequations: `<=~ <~ >~ >=~`

Constraint store

Constraints are not immediately taken into account and **must be added** to the **constraint store** with function:
`Cstr.post: Cstr.t -> unit`
No propagation occurs before *posting* the constraint

Propagation of Arithmetic Constraints

$xy - 2z \geq 90, \quad (x, y, z) \in [0, 10]^3$

The printing of constraints shows how equations can be internally rewritten with equivalent constraints (e.g. non-linear terms are represented with new variables):

```
# let [|x;y;z|] as vars = Fd.array 3 0 10;;
# let ineq =
    fd2e x *~ fd2e y -~ i2e 2 *~ fd2e z >=~ i2e 90;;
val ineq : Facile.Cstr.t = <abstr>
# Cstr.fprint stdout ineq;;
3: Fcl_linear.linear: +2._2[0..10] -1._3[0..100] <= -90
```

Bound-consistency with interval arithmetic

```
# Cstr.post ineq;;
# Cstr.fprint stdout ineq;;
3: Fcl_linear.linear: +2._2[0..5] -1._3[90..100] <= -90
# Fd.fprint_array stdout vars;;
[|_0[9..10]; _1[9..10]; _2[0..5]|]
```

Global Constraints

Global Constraints

- FaCiLe provides several **global constraints** taking **arrays of variables** as parameters
- Each global constraint (usually named `cstr`) can be accessed through its own module:
  - `Alldiff`: all different
  - `FdArray.get`: indexation (“element” constraint)
  - `FdArray.min,max`: extremum
  - `Gcc`: global cardinality
  - `Sort`: sorting

All Different Constraint

All Different

- Semantic for an array of size n: `Alldiff.cstr vars`

$$\texttt{vars.}(i) \neq \texttt{vars.}(j), \quad 0 \leq i \neq j < n$$

- The “all different” constraint is the most frequently used global constraints, notably to model:
  - a set of cardinal n with an array of size n (and not a multiset with possibly several occurrences of the same element)
  - a permutation of n elements (e.g. TSP)
  - a clique in a coloring problem
  - a Sudoku game...
- The constraint is in module `Alldiff`:

```
type algo = Lazy | Bin_matching of Var.Fd.event
val cstr : ?algo:algo -> Var.Fd.t array -> Cstr.t
```

Propagation of the All Different Constraint

Lazy (default)

- Equivalent to $\frac{n(n-1)}{2}$ disequations (but less overhead)
- Woken whenever a variable become assigned

Bin_matching event

- **Detects global failure** when no binary matching on the variable/value bipartite graph can be found
- Waking events can be finely tuned:

  `on_subst` a variable has been assigned
  `on_refine` a domain has been modified

  Events `on_min` and `on_max` are also available to waken constraints (not `Alldiff`) that propagate on bounds modification

Binary Constraint

Binary Constraint

Module `Binary` provides a **binary constraint** taking two variables and a list of integer couples:

- by default, the list of couples are the forbidden pairs (“nogoods”)
- optional parameter `~nogoods` can be set to `false` to take authorized pairs instead
- optional parameter `~algo` can be chosen among `Binary.AC3` or `Binary.AC6` (default)

```
# let [|x;y|] as vars = Fd.array 2 1 3;;
# let nogoods = [(1,1);(1,3);(2,1);(2,2);(2,3);(3,1)];;
# Cstr.post (Binary.cstr x y nogoods);;
# Fd.fprint_array stdout vars;;
[|_0[1;3]; _1[2..3]|]
```

Search Goal

Module Goal

- Search for a solutions performed with **goals** (cf. Prolog)
- Goals are not immediately executed but must be passed as the parameter to function `solve: t -> bool` which returns `true` if the goal **succeeds** or `false` whenever it **fails**
- A goal either succeeds or fails immediately, or returns a **subgoal**
- Goals can be combined with **conjunction** &&~ or **disjunction** ||~ (provided by module Easy)

Assignment goals

- `indomain: Fd.t -> t` non-deterministically assigns one variable, trying its values in increasing order
- `labeling: Fd.t array -> t` to assign an array of variables (provided by submodule `Goals.Array`)

Structure of a Constraint Program

1. Data processing
2. Definition of the variables
3. Posting of the constraints
4. Creation of an assignment goal
5. Search for solution(s) (possibly with optimization)
6. Printing of the solution(s)

Plan

Search Goals

Execution

The execution of goals is performed as in Prolog:

- **execution is deferred** until the goal is selected
- goals are selected in **left-to-right** order
- the alternative of a **choice point** is explored on **backtracking**, i.e. when the left goal of the disjunction fails

Module Goals

- Type of goals: `Goals.t`
- `Goals.fail` and `Goals.success` respectively **fails** and **succeeds** immediately
- Combined with **logical operators** and **iterators**
- Goals can be created to perform **side-effects** only
- Goals can return **subgoals**
- As in Prolog, goals can be **recursive**

Side-Effect Goals

Creation `atomic: (unit -> unit) -> t`

Creates a “side-effect goal” with **no subgoal**, **always succeeds**: e.g. printing, storing solution...

Printing all solutions

```
# let print_var = fun x ->
#   Goals.atomic
#     (fun () -> Printf.printf "%a\n" Fd.fprint x);;
val print_var : Fd.t -> Goals.t = <fun>
# let labprint = fun x ->
    Goals.indomain x &&~ print_var x &&~ Goals.fail;;
val labprint : Fd.t -> Goals.t = <fun>
# let x = Fd.interval ~name:"x" 1 3;;
# Goals.solve (labprint x);;
x[1]
x[2]
x[3]
- : bool = false
```


Side-Effect Goals

Storing all solutions: storage

```
let storage = fun vars ->
  let allsol = ref [] in
  let store =
    Goals.atomic
      (fun () ->
         let sol = Array.map Fd.elt_value vars in
         allsol := sol :: !allsol) in
  let get = fun () -> List.rev !allsol in
  (store, get)

(* int array printer *)
let fprint_sol = fun ch sol ->
  Printf.fprintf ch "[%d" sol.(0);
  for i = 1 to Array.length sol - 1 do
    Printf.fprintf ch ";%d" sol.(i) done;
  Printf.fprintf ch "]"
```

Side-Effect Goals

Storing all solutions: solving

```
let solve = fun n ->
  let vars = Fd.array n 1 n in
  let (store, get_allsol) = storage vars in
  Cstr.post (Alldiff.cstr vars);
  let goal = Goals.Array.labeling vars in
  ignore (Goals.solve (goal &&~ store &&~ Goals.fail));
  List.iter
    (fun sol -> Printf.printf "%a\n" fprint_sol sol)
    (get_allsol ())
let () = solve (int_of_string Sys.argv.(1))
$ ./allsol.out 3
[1;2;3]
[1;3;2]
[2;1;3]
[2;3;1]
[3;1;2]
[3;2;1]
```


Subgoals: Goals Returning Goals

Creation `create: ('a -> t) -> 'a -> t`

- Besides imperative processing or side-effects on variables executed within their body, goals may also return **subgoals**: e.g. non-deterministic assignment, logic combination of goals...
- `create g x` returns a goal that will apply `g` to `x` when it is executed. `g` must return a **subgoal** or `success` to terminate.

Splitting a domain in two (once)

```
let split = fun x ->
  if Fd.is_bound x then Goals.success else
  let (xmin, xmax) = Fd.min_max x in
  let mid = (xmin+xmax) / 2 in
  Goals.atomic (fun () -> Fd.refine_up x mid) ||~
  Goals.atomic (fun () -> Fd.refine_low x (mid+1))
let gsplit = Goals.create split
```

Recursive Goals

Creation `create_rec: (t -> t) -> t`

- As in Prolog, **recursive goals** can be defined
- `create_rec g` returns a goal that will apply g to the **goal itself** when it is executed. g must return a **subgoal** or `success` to terminate.
- Parameters are passed through **closure**

Recursive Goals

Non-deterministic assignment

```
let indomain = fun x ->
  Goals.create_rec
    (fun self ->
       if Fd.is_bound x then Goals.success else
       let xmin = Fd.min x in
       Goals.unify x xmin ||~
       (Goals.atomic (fun () -> Fd.remove x xmin)
        &&~ self))
```

Recursive dichotomic search

```
let dicho = fun var ->
  Goals.create_rec
    (fun self ->
      if Fd.is_bound var then Goals.success else
      split var &&~ self)
```

Recursive Goals

Dichotomic search with printing

```
let dicho_print = fun var ->
  Goals.create_rec
    (fun self ->
      Printf.printf "%a\n" Fd.fprint var;
      if Fd.is_bound var then Goals.success else
      split var &&~ self)
let () =
  let x = Fd.interval ~name:"x" 1 8 in
  ignore (Goals.solve (dicho_print x &&~ Goals.fail))
```

```
x[1..8]          x[3..4]          x[5]
x[1..4]          x[3]             x[6]
x[1..2]          x[4]             x[7..8]
x[1]             x[5..8]          x[7]
x[2]             x[5..6]          x[8]
```


Logic Operators and Assignment Goals

Operators

- &&~ conjunction
- ||~ disjunction

Assignment goal

`unify: Fd.t -> Fd.elt -> t`
assigns a given value to a variable

Non-deterministic assignment

- `indomain: Fd.t -> t` in increasing order
- `random: Fd.t -> t` in random order
- `instantiate: (Domain.t->Domain.elt)->Fd.t->t` order provided by a user function
- `dichotomic: ?order:order -> Fd.t -> t` binary search with *order* being `Incr` (default) or `Decr`

Assignment Goals

Assignment of a boolean variable in *decreasing* order

```
# let booldecrease = Goals.instantiate (fun _ -> 1);;
# let x = Fd.interval 0 1;;
# Goals.solve
    (booldecrease x &&~ print_var x &&~ Goals.fail);;
_3[1]
_3[0]
- : bool = false
```


Iterators

Submodule Goals.Array

- Arrays can be of **any type**
- **Assignment** for array of variables
- Generalized **conjunction** (`forall`) and **disjunction** (`exists`)
- An optional argument `~select` determines the goal **ordering**, i.e. the **search strategy** for assignment goals

```
forall: ?select:('a array -> int)
          -> ('a -> t) -> 'a array -> t
```

- With the index:

```
foralli: ?select:('a array -> int)
           -> (int -> 'a -> t) -> 'a array -> t
```

`Goals.Array.labeling` is defined as

```
let labeling = Goals.Array.forall Goals.indomain
```

Search strategy

Minimal domain size and maximal “degree”

```
let dom_deg =
  Goals.Array.choose_index
    (fun v1 v2 ->
      let s1 = Fd.size v1 and s2 = Fd.size v2 in
      s1 < s2 ||
      (s1 = s2 &&
      Fd.constraints_number v1 > Fd.constraints_number v2))

let goal = fun vars ->
  Goals.Array.forall ~select:dom_deg Goals.indomain vars
```

Weighted degree

```
let wdeg_order =
  Goals.Array.choose_index
    (fun a1 a2 -> Fd.wdeg a1 > Fd.wdeg a2)
```


Optimization

Minimization goal

- Goals can be solved by **Branch & Bound** to optimize a **cost**
- `minimize : t -> Fd.t -> (int -> unit) -> t` `Goals.minimize g c sol` returns a minimization goal of goal `g` for cost `c`, calling function `sol` each time a new solution is found with the value of the cost as argument
- For **maximization**, the opposite of the cost should be passed to `minimize`
- Goal `g` **must assign the cost** `c` (which is obviously the case when the cost is defined as a function of the decision variables that the goal assigns)
- Function `sol` can be used to print or store the last solution
- An optimization goal always **fails** as the optimality proof is obtained when the search space is exhausted

Optimization

Give change with 1, 2, 5, 10 and 20 € coins

Minimizing the number of coins

```
let sum = 123 in
let values = [|1;2;5;10;20|] in
let nb_coins = Fd.array 5 0 sum in
Cstr.post
  (Arith.scalprod_fd values nb_coins =~ i2e sum);
let cost = Arith.e2fd (Arith.sum_fd nb_coins) in
let labeling = Goals.Array.labeling nb_coins in
let solution = fun c ->
  Printf.printf "nb_coins:%a cost:%d\n"
    Fd.fprint_array nb_coins c in
let opti = Goals.minimize labeling cost solution in
ignore (Goals.solve opti)
```

```
nb_coins:[|_0[0]; _1[4]; _2[1]; _3[1]; _4[5]|] cost:11
nb_coins:[|_0[1]; _1[1]; _2[0]; _3[0]; _4[6]|] cost:8
```


Changing the Search Strategy

Assigning the biggest coins first

Before the search, the arithmetic constraint has propagated:

```
[|_0[0..123]; _1[0..61]; _2[0..24]; _3[0..12]; _4[0..6]|]
```

We could start with the biggest coins, which have the smallest domain, to obtain the optimal solution more efficiently:

```
let dom_order =
  Goals.Array.choose_index
    (fun v1 v2 -> Fd.size v1 < Fd.size v2)
[...]
let labeling =
  Goals.Array.forall
    ~select:dom_order Goals.indomain nb_coins in
[...]
```

Changing the Search Strategy

```
nb_coins:[|_0[123]; _1[0]; _2[0]; _3[0]; _4[0]|] cost:123
nb_coins:[|_0[121]; _1[1]; _2[0]; _3[0]; _4[0]|] cost:122
nb_coins:[|_0[119]; _1[2]; _2[0]; _3[0]; _4[0]|] cost:121
nb_coins:[|_0[117]; _1[3]; _2[0]; _3[0]; _4[0]|] cost:120
nb_coins:[|_0[115]; _1[4]; _2[0]; _3[0]; _4[0]|] cost:119
[...]
nb_coins:[|_0[1]; _1[1]; _2[0]; _3[10]; _4[1]|] cost:13
nb_coins:[|_0[1]; _1[1]; _2[0]; _3[8]; _4[2]|] cost:12
nb_coins:[|_0[1]; _1[1]; _2[0]; _3[6]; _4[3]|] cost:11
nb_coins:[|_0[1]; _1[1]; _2[0]; _3[4]; _4[4]|] cost:10
nb_coins:[|_0[1]; _1[1]; _2[0]; _3[2]; _4[5]|] cost:9
nb_coins:[|_0[1]; _1[1]; _2[0]; _3[0]; _4[6]|] cost:8
```

- By default, `Goals.indomain` tries to assign variables in increasing order of their values
- So we must change the value ordering as well to try the maximal value first

Changing the Search Strategy

Trying maximal value first

```
let indomain_max =
  Goals.instantiate (fun d -> Domain.max d)
[...]
let labeling =
  Goals.Array.forall
    ~select:dom_order indomain_max nb_coins in
[...]
```

The optimal solution is then found immediately:

```
nb_coins:[|_0[1]; _1[1]; _2[0]; _3[0]; _4[6]|] cost:8
```

Optimization

Generalization of the problem

As few coins as possible to always be able to give change for each sum $\in [1, sum\_max]$ for any set of coins:

```
let coins = fun values sum_max ->
  let n = Array.length values in
  let mat =
    Array.init sum_max
      (fun sum -> Fd.array n 0 (sum+1)) in
  let tmat =
    Array.init n
      (fun j ->
         Array.init sum_max (fun i -> mat.(i).(j))) in
  let nb_min_coins = Array.map FdArray.max tmat in
  for sum = 1 to sum_max do
    Cstr.post
      (Arith.scalprod_fd values mat.(sum-1) =~ i2e sum)
  done;
  let cost = Arith.e2fd (Arith.sum_fd nb_min_coins) in
```


Optimization

Generalization of the problem

```
  let vars = Array.concat (Array.to_list mat) in
  let labeling = Goals.Array.labeling vars in
  let solution = fun c ->
    Printf.printf "%a cost:%d\n"
      Fd.fprint_array nb_min_coins c in
  let opti = Goals.minimize labeling cost solution in
  ignore (Goals.solve opti)
let () =
  let values = [|1;2;5;10;20|] in
  let sum_max = 100 in
  coins values sum_max
```

```
[|_500[1]; _501[4]; _502[1]; _503[1]; _504[5]|] cost:12
[|_500[1]; _501[4]; _502[1]; _503[1]; _504[4]|] cost:11
[|_500[1]; _501[3]; _502[1]; _503[1]; _504[4]|] cost:10
[|_500[1]; _501[2]; _502[1]; _503[1]; _504[4]|] cost:9
```

Plan

Three Models for Eight Queens

N-queens problem

- Place 8 (n) queens on a chessboard so that no two queens attack each other, i.e. no two queens are on the same row, column or diagonal
- Not in NPC but often used as a benchmark

Different modeling

Different:
- domains
- number of variables
- search space sizes
- constraints
- search strategies

which lead to different:
- performances
- sizes of solvable instances

Booleans

$n \times n$ boolean variables

Constraints:

- n queens only
- no attack on rows, columns and diagonals

```
let vars = Fd.array (n*n) 0 1 in
Cstr.post (Arith.sum_fd vars =~ i2e n);
for i = 0 to n-1 do
 for j = 0 to n-1 do
  for k = 1 to n-i-1 do
   Cstr.post
    (fd2e vars.(i*n+j) +~ fd2e vars.((i+k)*n+j) <~ i2e 2)
   done;
   for k = 1 to n-j-1 do
    Cstr.post
     (fd2e vars.(i*n+j) +~ fd2e vars.(i*n+(j+k)) <~ i2e 2)
   done;
```


Booleans

```
  for k = 1 to min (n-i-1) (n-j-1) do
   Cstr.post
    (fd2e vars.(i*n+j)+~fd2e vars.((i+k)*n+(j+k)) <~ i2e 2)
  done;
  for k = 1 to min (n-i-1) j do
   Cstr.post
    (fd2e vars.(i*n+j)+~fd2e vars.((i+k)*n+(j-k)) <~ i2e 2)
  done done done;
let goal = Goals.Array.labeling vars in
let nb_cstr = List.length (Cstr.active_store ()) in
let bt = ref 0 in
let control = fun b -> bt := b in
let start = Sys.time () in
if Goals.solve ~control goal then begin
  let duration = Sys.time () -. start in
  Printf.printf "bt: %d\nnb cstr: %d\ntime: %g\n"
    !bt nb_cstr duration end
else Printf.printf "No solution found\n"
```

Booleans

Results

- Many variables, constraints and failures
- Huge search space

```
bt: 8540
nb cstr: 729
time: 2.364
```


Integer Couples

Coordinates

- **Structure** of the pb. poorly **modeled** by the boolean CSP
- Few variables are assigned to 1 (true)
- Represent queens positions only:
  - $2 \times n$ variables
  - no attack on rows, columns and diagonals

```
let xs = Fd.array n 0 (n-1) in
let ys = Fd.array n 0 (n-1) in
for i = 0 to n-1 do
 for j = i+1 to n-1 do
  Cstr.post (fd2e xs.(i) <>~ fd2e xs.(j));
  Cstr.post (fd2e ys.(i) <>~ fd2e ys.(j));
  Cstr.post
   (fd2e xs.(j)-~fd2e xs.(i) <>~ fd2e ys.(j)-~fd2e ys.(i));
  Cstr.post
   (fd2e xs.(i)-~fd2e xs.(j) <>~ fd2e ys.(j)-~fd2e ys.(i));
[...]
```

Integer Couples

Results

- Much smaller search space
- Less variables and constraints
- Less backtracks
- 100 times faster

```
bt: 24
nb cstr: 112
time: 0.016
```


Single Integers

Position in each row

The results of the former model:

```
0 1 2 3 4 5 6 7
0 4 7 5 2 6 1 3
```

suggest to place only one queen by row:

- n variables
- no attack on columns and diagonals

```
let ys = Fd.array n 0 (n-1) in
for i = 0 to n-1 do
 for j = i+1 to n-1 do
  Cstr.post (fd2e ys.(i) <>~ fd2e ys.(j));
  Cstr.post (fd2e ys.(j) -~ fd2e ys.(i) <>~ i2e (j-i));
  Cstr.post (fd2e ys.(j) -~ fd2e ys.(i) <>~ i2e (i-j));
[...]
```

Single Integers

Results

- Smaller search space
- Less variables and constraints
- Faster

```
bt: 24
nb cstr: 84
time: 0.012
```


Global Constraint

All different

- The no-attack constraints on columns imply that all *ys* are distinct
- A **global constraint**, more efficient, can replace them

```
let algo = Alldiff.Bin_matching Fd.on_subst in
Cstr.post (Alldiff.cstr ~algo ys);
for i = 0 to n-1 do
 for j = i+1 to n-1 do
  Cstr.post (fd2e ys.(j) -~ fd2e ys.(i) <>~ i2e (j-i));
  Cstr.post (fd2e ys.(j) -~ fd2e ys.(i) <>~ i2e (i-j))
[...]
```

Results

Fewer constraints even if same search space, comparable efficiency (resp. 22 s and 14 s for 20 queens)

```
bt: 24    nb cstr: 58    time: 0.012
```

Fewer Constraints, More Variables

Rewriting of the disequations

$$j - i \neq ys[j] - ys[i] \iff ys[i] - i \neq ys[j] - j$$
$$i - j \neq ys[j] - ys[i] \iff ys[i] + i \neq ys[j] + j$$

Introducing auxiliary variables can simplify the constraints statement

```
let ys1 =
 Array.mapi
  (fun i ysi -> Arith.e2fd (fd2e ysi -~ i2e i)) ys in
let ys2 =
 Array.mapi
  (fun i ysi -> Arith.e2fd (fd2e ysi +~ i2e i)) ys in
let algo = Alldiff.Bin_matching Fd.on_subst in
Cstr.post (Alldiff.cstr ~algo ys);
Cstr.post (Alldiff.cstr ~algo ys1);
Cstr.post (Alldiff.cstr ~algo ys2);
```


Fewer Constraints, More Variables

Global constraints

A global constraint (generally) **propagates more** than a set of equivalent simpler constraints

Results for 20 queens (in native code)

Without ys1 and ys2:

```
bt: 30707
nb cstr: 382
time: 0.576
```

With all-different constraints (Bin_matching):

```
bt: 30566 (on_subst) 30440 (on_refine)
nb cstr: 46
time: 0.748 (on_subst) 0.792 (on_refine)
```

With Lazy:

```
bt: 37320
nb cstr: 43
time: 0.66
```

Search Strategy

Two kinds of choice during the search phase

- variable assignment ordering
- value selection ordering

The default strategy:

- first non-assigned variable
- values by increasing order

For 20 queens

```
bt: 30440
nb cstr: 46
time: 0.8
```


“First-Fail” Principle

Variable ordering

One of the most common strategy is to choose the variable with smallest domain first

```
let dom_order =
 Goals.Array.choose_index
  (fun v1 v2 -> Fd.size v1 < Fd.size v2)
[...]
let goal =
 Goals.Array.forall
  ~select:dom_order Goals.indomain ys in
```

Much more efficient for 20 queens

```
bt: 30
nb cstr: 46
time: 0.004
```

Even Better for the Queens

Second criterion

- Smallest domain as first criterion
- Smallest minimum as second one

```
let dom_min_order =
  Goals.Array.choose_index
    (fun v1 v2 ->
      let s1 = Fd.size v1 and s2 = Fd.size v2 in
      s1 < s2 || (s1 = s2 && Fd.min v1 < Fd.min v2))
```

Results

For 20 queens:

```
bt: 25
nb cstr: 46
time: 0.004
```

And for a thousand and one queens...

```
bt: 0
nb cstr: 2008
time: 9.48
```


Plan

All Different

Optimal Golomb Ruler (OGR)

A Golomb ruler of order n is a set of n marks at integer positions on a ruler such that all distances between any two marks are different:

$$\forall i < j, k < l, (i,j) \neq (k,l) \quad |x_j - x_i| \neq |x_l - x_k|$$

Symmetries:

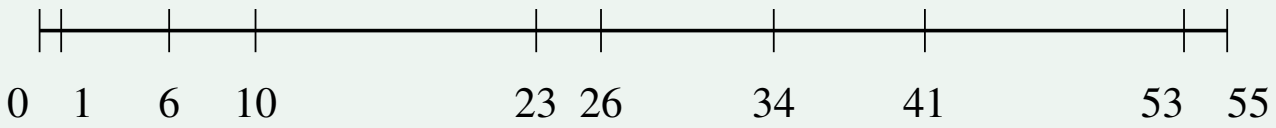
- permutation of the variables: $x_1 < x_2 < \cdots < x_n$
  the constraints can then be written without absolute values
- translation: $x_1 = 0$
- reflection: $x_2 - x_1 < x_n - x_{n-1}$

A Golomb ruler is *optimal* if it is the shortest of a given order

OGR are known up to order 27 only.
Order 27 was proved optimal in 2014 after 5 years of computing...

All Different

OGR of order 10



Performances

size	on_refine		on_subst		Lazy		$\Theta(n^4)$ $\neq$	
	CPU	bt	CPU	bt	CPU	bt	CPU	bt
5	0.00	10	0.00	13	0.00	15	0.00	15
6	0.02	34	0.02	38	0.02	55	0.02	55
7	0.04	227	0.04	240	0.07	323	0.13	324
8	0.40	1416	0.52	1537	0.61	2128	1.68	2124
9	3.87	8383	4.17	9667	4.85	13964	17.82	13956
10	24.79	34121	27.70	44098	39.60	79117	169.24	79132

Global Cardinality

Module Gcc: Global Cardinality Constraint

```
type level = Basic | Medium | High
cstr : ?level:level -> Fd.t array -> (Fd.t * int) array
         -> Cstr.t
```

`Gcc.cstr vars cardvals` returns a constraint ensuring that, for each couple (c_j, v_j) of array `cardvals`, the value v_j appears c_j times in variables array `vars`:

$$\forall j \quad c_j = \left|\{x_i = v_j, \forall i \in [1, n]\}\right|$$

- Three levels of propagation are provided
- The propagation is based on the Ford-Fulkerson algorithm which computes a maximal flow in a graph with capacities
- It is a generalization of the all-different constraint

Global Cardinality

Magic Sequence

A *magic sequence* is a sequence $x_0, x_1, \ldots, x_n$ of numbers such that 0 appears x_0 times in the sequence, 1 appears x_1 times, ..., i appears x_i times...

```
let vars = Fd.array (n+1) 0 n in
let card_vals = Array.mapi (fun i x -> (x, i)) vars in
Cstr.post (Gcc.cstr ~level:Gcc.Medium x card_vals);
(* redundant constraint *)
let vals = Array.init (n+1) (fun i -> i) in
Cstr.post (Arith.scalprod_fd vals vars =~ i2e (n+1));
...
```

Global Cardinality

ATM: runway/sector with hourly capacity

- Capacity is fixed as the maximal number of a/c per hour
- Can change every hour
- Delay takeoffs to respect the capacities

```
let delays = Fd.array n 0 maxdelay in
let hours =
  Array.mapi
    (fun i di ->
       let h = (i2e takeoffs.(i) +~ fd2e di) /~ i2e 60 in
       Arith.e2fd h)
    delays in
Array.iter (fun h -> Cstr.post (fd2e h <~ i2e 24)) hours;
let cardvals =
  Array.mapi (fun h ch -> (Fd.interval 0 ch, h)) capa in
Cstr.post (Gcc.cstr hours cardvals);
```


Sorting Constraint

Module Sorting

`val sort : Fd.t array -> Fd.t array`

`Sorting.sort vars` returns an array of variables constrained to map to the variables of `vars` in increasing order

```
# let vars = Fd.array 3 1 5;;
# let sorted = Sorting.sort vars;;
# Cstr.post (fd2e vars.(2) <~ i2e 3);;
vars:[|_0[1..5]; _1[1..5]; _2[1..2]|]
sorted:[|_3[1..2]; _4[1..5]; _5[1..5]|]
# Cstr.post (fd2e vars.(1) >~ i2e 3);;
vars:[|_0[1..5]; _1[4..5]; _2[1..2]|]
sorted:[|_3[1..2]; _4[1..5]; _5[4..5]|]
# Cstr.post (fd2e sorted.(1) =~ i2e 2);;
vars:[|_0[1..2]; _1[4..5]; _2[1..2]|]
sorted:[|_3[1..2]; _4[2]; _5[4..5]|]
# Cstr.post (fd2e vars.(0) <~ fd2e vars.(2));;
vars:[|_0[1]; _1[4..5]; _2[2]|]
sorted:[|_3[1]; _4[2]; _5[4..5]|]
```

Sorting Constraint

ATM: continuous capacity

Dual model:

- With `Sorting`, we can use the order of entry of a/c
- Instead of limiting the number of a/c in each hour, we space out the i^{th} and $(i+c)^{\text{th}}$ a/c with at least one hour:

$$t_{i+c} - t_i > period$$

```
let starts =
 Array.mapi
  (fun i di -> Arith.e2fd (fd2e di +~ i2e takeoffs.(i)))
  delays in
Array.iter
 (fun si -> Cstr.post (fd2e si <~ i2e 1440)) starts;
let sorted = Sorting.sort starts in
for i = 0 to n - 1 - c do
 Cstr.post (fd2e sorted.(i+c) -~ fd2e sorted.(i) >~ i2e 60)
done;
```


Array Constraints

Module FdArray

- `min|max vars` returns the minimal or maximal value of `vars`
- `get vars idx` returns a variable `y = vars.(idx)`, `idx` being a variable itself (“indexation” constraint)

```
# let t =
    [|Fd.interval 7 12;Fd.interval 2 5;Fd.interval 4 8|];;
# let i = Fd.interval (-10) 10;;
# let ti = FdArray.get t i;;
i:_3[0..2] ti:_4[2..12]
# Cstr.post (fd2e i >~ i2e 0);;
i:_3[1..2] ti:_4[2..8]
# Cstr.post (fd2e ti <~ i2e 4);;
i:_3[1] ti:_4[2..3] t:[|_2[7..12]; _1[2..3]; _0[4..8]|]
# let mint = FdArray.min t;;
mint:_5[2..3]
```

Plan

Constraint Reification (meta-constraint, soft constraint)

Module `Reify`

- Instead of being enforced directly, some constraints can be:
  - combined with **logical operators** to build new constraints
  - or considered as a **boolean variable** (0 if the constraint fails, 1 if it succeeds) and used in arithmetic constraints
- Not all constraints are reifiable (cf. documentation): a **check** and a **negation** function must be defined
- `Reify.boolean cstr` returns a boolean variable

Logical operators

- Provided by module Easy: $\wedge$ $\vee$ $\Rightarrow$ $\Leftrightarrow$
  `&&~~` `||~~` `=>~~` `<=>~~`
- XOR and negation in module `Reify`:
  `xor : Cstr.t -> Cstr.t -> Cstr.t` `not : Cstr.t -> Cstr.t`

Constraint Reification

Arithmetic Reification

- To ease the writing of reified arithmetic expressions, specific operators are provided by `Arith` (and `Easy`):

  `<~~  <=~~  =~~  >=~~  >~~  <>~~`

- With type: `Arith.t -> Arith.t -> Arith.t`
- `e1 op~~ e2  ≡  fd2e (Reify.boolean (e1 op~ e2))`

Global cardinality constraint with reifications

```
let gcc = fun vars cardvals ->
 Array.iter
  (fun (c, v) ->
   let iseqv = Array.map (fun var -> var =~~ i2e v) vars in
   Cstr.post (Arith.sum iseqv =~ fd2e c))
  cardvals;
 let n = Array.length vars in (* redundant constraint *)
 Cstr.post (Arith.sum_fd (Array.map fst cardvals) =~ i2e n)
```


Application: Scheduling

Scheduling with Resource Allocation

For a set of tasks to be executed and a set of technicians, choose a technician for each task and set the start times of the tasks:

- Data:
  - d_i duration of task i
  - tr_{ij} minimal time interval between the end of task i and the start of task j if they're executed by the same technician
- Variables:
  - x_i technician performing task i
  - t_i start time of task i
- Constraints: precedence constraints on all pairs $i < j$ of tasks if the same technician performs both i and j

$$\forall i < j, \quad x_i = x_j \Rightarrow (t_i + d_i + tr_{ij} \leq t_j \ \vee \ t_j + d_j + tr_{ji} \leq t_i)$$

Application: Scheduling

With arithmetic reifications and “big M”

```
let bigm = max_int / 3 in
let b1 = Fd.interval 0 1 and b2 = Fd.interval 0 1 in
Cstr.post
  (fd2e t.(i) +~ i2e (d.(i) + tr.(i).(j))
     <=~ fd2e t.(j) +~ fd2e b1 *~ i2e bigm);
Cstr.post
  (fd2e t.(j) +~ i2e (d.(j) + tr.(j).(i))
     <=~ fd2e t.(i) +~ fd2e b2 *~ i2e bigm);
Cstr.post
  ((fd2e x.(i) =~~ fd2e x.(j)) +~ fd2e b1 +~ fd2e b2 =~ i2e 2);
```

- b1 allows to relax the precedence constraint between i and j and b2 between j and i
- If the two tasks are executed by distinct technicians, both precedence constraints are relaxed. Otherwise, one of them must be satisfied (b1=0 or b2=0)
- b1 and b2 must be assigned by the search goal

Application: Scheduling

With logical operators

The model can also be stated with an implication (=>~~) and a disjunction (||~~):

```
Cstr.post
 ((fd2e x.(i) =~ fd2e x.(j)) =>~~
  ((fd2e t.(i) +~ i2e (d.(i)+tr.(i).(j)) <=~ fd2e t.(j))
     ||~~
   (fd2e t.(j) +~ i2e (d.(j)+tr.(j).(i)) <=~ fd2e t.(i))))
```

Plan

User-Defined Constraints

New Constraints

- When the constraint language of a solver is not rich enough
- If additional (redundant) propagation is needed (and possible) on a specific feature of the problem
- New constraints can be defined by the user

Two functions at least must be specified

- An `update` function that **propagates** the constraint by filtering the domains to remove inconsistent values
- A `delay` function specifying the **waking events**

To be reifiable, two additional functions are needed

- A `check` function that **checks** if the constraint is satisfied, violated or yet unknown **without modifying domains**
- A `not` function returning the **negation** of the constraint

Domain Filtering: module Fd

Domain access

```
is_var|is_bound : t -> bool
type ('a, 'b) concrete = Val of 'a | Unk of 'b
value : t -> (domain, elt) concrete
dom : t -> domain
min|max : t -> elt
member : t -> elt -> bool
```

Domain Modification

```
unify : t -> elt -> unit
refine : t -> domain -> unit
refine_low : t -> elt -> unit
refine_up : t -> elt -> unit
```

More set operations are available in module `Domain`

```
included|disjoint : t -> t -> bool
smallest_geq|greatest_leq : t -> elt -> elt
intersection|union|diff : t -> t -> t
```


Creation

Module `Cstr`

```
create : ?check:(unit -> bool) -> ?not:(unit -> t) ->
         (int -> bool) -> (t -> unit) -> t
```

- `create update delay` returns a new constraint that will call `update` whenever the events specified in `delay` occurs
- Optional arguments `check` and `not` must be provided to create a **reifiable** constraint

`update : int -> bool`

This function should perform **propagation**, i.e. domains filtering and consistency checks. It must:

- return `true` iff the constraint is consistent
- **raise** exception `Stak.Fail` if an inconsistency is detected
- return `false` otherwise

Its integer parameter should be ignored (if *waking ids* are not used)

Events and Scheduling

`delay : t -> unit`

- This function schedules the awakening of the constraint by calling `Fd.delay: event list -> t -> Cstr.t -> unit`
- It takes the constraint itself as parameter to be used as argument of `Fd.delay` (for technical reasons)
- `Fd.delay events x ct` schedules constraint `ct` to be waken when one of the events in list `events` occurs

Events

Four events are defined by module `Fd`:

- `on_subst`: assignment
- `on_min|on_max`: modification of the minimal/maximal value
- `on_refine`: any modification of the domain

Constraints scheduled on general events are waken when more specific events occur

Example: the Difference Constraint

Simple Constraint (not reifiable)

- The constraint is scheduled to propagate on **assignment** of one of the two variables
- When one of the variables is assigned, its value is removed from the domain of the other one

```
let cstr = fun x y ->
  let name = "different" in
  let update = fun _ ->
    if Fd.is_bound x then
      begin Fd.remove y (Fd.elt_value x); true end
    else if Fd.is_bound y then
      begin Fd.remove x (Fd.elt_value y); true end
    else false in
  let delay = fun ct ->
    Fd.delay [Fd.on_subst] x ct;
    Fd.delay [Fd.on_subst] y ct in
  Cstr.create ~name update delay
```

Example: the Difference Constraint

Reifiable Constraint

- `check ()` must return `true` is the constraint is **satisfied**, `false` if it is **violated** and raise exception `Cstr.DontKnow` if it is yet **undetermined**
- The domains must **not be modified**
- `not ()` must return the **negation** of the constraint
- If the **boolean variable** associated with the reification is set to 1, the constraint is **posted**, and if it is set to 0, the **negation** of the constraint is posted
- Conversely, if the constraint is satisfied, the boolean is set to 1, and to 0 if it is violated

Example: the Difference Constraint

Reification functions

- `check`:
  - `true` if domains are disjoint
  - `false` if variables are assigned the same value
  - `DontKnow` otherwise
- `not`: $\overline{x \neq y} \equiv x = y$

```
let check = fun () ->
  match (Fd.value x, Fd.value y) with
    (Val a, Val b) -> a <> b
  | (Val a, Unk dy) when not (Domain.member dy a) -> true
  | (Unk dx, Val b) when not (Domain.member dx b) -> true
  | (Unk dx, Unk dy) when
      Domain.is_empty (Domain.intersection dx dy) -> true
  | _ -> raise Cstr.DontKnow in
let not = fun () -> fd2e x =~ fd2e y in
Cstr.create ~name ~check ~not update delay
```