FaCiLe: A Functional Constraints Library
Release beta-0.1

N. Barnier P. Brisset

January 26, 2001

Preface

FaCiLe is a constraint programming library over integer finite domain written in OCaml[4]. It offers all usual facilities to create and manipulate finite domain variables, arithmetic expressions and constraints (possibly non-linear), built-in global constraints and search goals. FaCiLe allows as well to build easily user-defined constraints and goals (including recursive ones), making pervasive use of OCaml higher-order functionals to provide a simple and flexible interface for the user. As FaCiLe is an OCaml library and not yet another language, the user benefits from type inference and strong typing discipline, high level of abstraction, modules and objects system, as well as native code compilation efficiency, garbage collection and replay debugger, all features of OCaml (among many others) that allow to prototype and experiment quickly: modeling, data processing and interface are implemented with the same powerful and efficient language.

This is not a document about constraint programming techniques but only a tool description.

It is not a course of functional programming

No overloading

No syntactic sugar hiding bugs

Not a race for the smallest number of lines for toy examples

same language for everything, preprocessing graphic interface (labltk, lablgtk)

Foreword

Portability

FaCiLe requires only the OCaml system (release 3.00 or greater) and should work in any environment supporting this system. It is developed in a Linux environment on PC architecture but does not use any specificities of Unix. It should work on other operating systems (i.e. MS Windows, Mac OS, ...), provided that the installation process is customised to the environment.

FaCiLe Structure and Naming Conventions

The library is splited into numerous modules and submodules. They are all included into the main module `Facile` which should be opened by any other modules using FaCiLe. All the modules are extensively described in part II of this documentation. We do not recommend to users to open modules in `Facile` but to use prefixed notations (e.g. function `post` of `Cstr` is written `Cstr.post`). Pseudo-module `Easy` is the exception and should be opened: it provides several aliases to most frequently used values (see 4.14).

To avoid interferences with other modules of the user, all the modules are aliased in the `Facile` module and implementation module files are all prefixed by `fcl_` (except of course `Facile` itself). For example, implementation of module `Gcc` is in file `fcl_gcc.ml` and alias

```
module Gcc = Fcl_gcc
```

is defined in `Facile` (`facile.ml`). This alias mechanism is entirely transparent to the user of FaCiLe except for the one interested by the implementation of the library. The only possible visibility of `Fcl_` prefix is given by the uncaught exceptions printer (e.g. `Fcl_stak.Fail` instead of `Stak.fail`).

The reference part of this documentation is automatically generated from module interfaces (`.mli`). Some available functions, types or modules are intentionally not documented or even hidden in `Facile` module. They are not intented to the casual user.

Values and types names try to benefit as much as possible from the modularity. For example, most of the types are named `t`: type of constraints is `Cstr.t`, type of domains is `Domain.t` ... In the same way, printing functions are named `fprint`, constraints are named `cstr` (e.g. `Gcc.cstr`), ...

Standard or *label* mode of the OCaml compiler (option `-labels`) may be used with the library. FaCiLe makes use of labels (labelled arguments) as less as possible; only optional arguments are labelled.

Compilation with FaCiLe

FaCiLe is provided as bytecode and native code[1] libraries.

Bytecode version is compiled with debugging information (`-g` option of `ocamlc`) and then can be used with the source-level replay debugger (`ocamldebug`). A lots of checks are done in this

[1] If supported by your architecture. See `http://caml.inria.fr/ocaml/portability.html`

mode and exceptions may be raised revealing bad usage of the system ("fatal" errors) or bugs in the system itself ("internal" errors). In the second case, diligent users should send a bug report to the developers.

In the native code version, these redundant checks are not done and this mode should be used only on well-tried code.

The `Makefile` in the `examples` directory of the distribution provides generic rules to compile with FaCiLe in both modes producing `.out` (bytecode) or `.opt` (native code) executables.

The library may also be used through linked toplevel produced with the following command (after installation):

```
ocamlmktop -o facile facile.cma
```

This is the toplevel used in the inlined examples of this documentation.

Availability

The FaCiLe distribution and documentation are available by anonymous FTP at:

`ftp://ftp.recherche.enac.fr/pub/facile`

There is also a web site for FaCiLe where general information can be found:

`http://www.recherche.enac.fr/opti/facile`

Questions, bug reports, ..., can be mailed to

`facile@recherche.enac.fr`

Installation

Installation of FaCiLe is described in the `README` file of the distribution. Here follows a copy of the corresponding part:

```
INSTALLATION:

  All you need is the Objective Caml 3.00 (or greater) compiler and
standard Unix tools (make, ...).

  0) Default installation put the library files (facile.cma,
facile.cmxa, facile.a facile.cmi) in the Ocaml library directory
(returned by "ocamlc -v"). If you want to choose another installation
directory, run the following command, else skip to the next step.

make FACILEDIR=<directory> configure

  1) First compile the library with a simple

        make

  2) Then install the library with a (usually as root)

        make install

  3) Check the installation
```

```
make check
```

```
You should get a solution for the 8 queens problem.
```

Examples

The directory `examples` of the distribution contains some examples and a generic `Makefile` to compile files with FaCiLe.

Examples are taken from the classic litterature:

Queens Place queens on a chessboard

Golf Organise a golf tournament for 8 teams of 4 players

Magic To count and to be counted

Marriage Stabilise preferences among spouses

Tiles To tile small squares in a big square

Contents

Part I

User's Manual

Chapter 1

Getting Started

This first chapter introduces the overall framework of FaCiLe and gives a preliminary insight about its programming environment and functionalities.

OCaml code using FaCiLe facilities (file `csp.ml` in the following examples) must be compiled with the library of object byte code `facile.cma` when batch compiling with `ocamlc`:

```
ocamlc facile.cma csp.ml
```

and with the library of object code `facile.cmxa` for native compilation with `ocamlopt`:

```
ocamlopt facile.cmxa csp.ml
```

provided that the standard installation of FaCiLe (and previously of the OCaml system of course) has been performed (see) and that the `facile.cm[x]a` files has been thereafter successfully created in the OCaml standard library directory. For larger programs, a generic Makefile can be found in directory `examples` (see).

It may however be convenient to use an OCaml custom toplevel to experiment toy examples or check small piece of serious (thus large) code. A FaCiLe toplevel (i.e. in which `facile.cma` is pre-loaded) is easily build with the following command:

```
ocamlmktop -o facile facile.cma
```

and invoked with `./facile`.

The two following sections give a quick overview of the main basic concepts of FaCiLe with the help of two very simple examples which are precisely explained step by step.

1.1 Basics

We first give a slight taste of FaCiLe with the recurrent trivial problem of the Canadian flag: one have to repaint the Canadian flag (shown in figure 1.1) with its two original colors, red and white, such that two neighbouring areas don't have the same color and the maple leaf is of course red. The CSP model is desperately straightforward:

- one variable for each area l, c, r and m;
- all variables have the same domain [0..1], 0 being red and 1, white;
- one difference constraint for each adjacence $l \neq c$, $c \neq r$, $m \neq c$ and the maple leaf is forced to be red $m = 0$.

The following piece of code solves this problem:

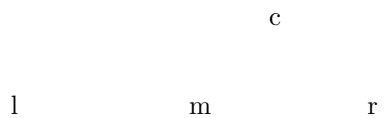

Figure 1.1: The problem of the Canadian flag

maple.ml

```
open Facile
open Easy
let _ =
  (* Variables *)
  let red = 0 and white = 1 in
  let dom = Domain.create [red; white] in
  let l = Fd.create dom and c = Fd.create dom
  and r = Fd.create dom and m = Fd.create dom in
  (* Constraints *)
  Cstr.post (fd2e l <>~ fd2e c);
  Cstr.post (fd2e c <>~ fd2e r);
  Cstr.post (fd2e m <>~ fd2e c);
  Cstr.post (fd2e m =~ i2e red);
  (* Goal *)
  let var_list = [l;c;r;m] in
  let goal = Goals.GlList.labeling var_list in
  (* Search *)
  if Goals.solve goal then begin
    Printf.printf "l="; Fd.fprint stdout l;
    Printf.printf " c="; Fd.fprint stdout c;
    Printf.printf " r="; Fd.fprint stdout r;
    Printf.printf " m="; Fd.fprint stdout m;
    print_newline () end
  else
    prerr_endline "No solution"
```

```
unix% ocamlc facile.cma maple.ml
unix% ./a.out
l=0 c=1 r=0 m=0
```

The new flag is therefore a faithful copy of the genuine one.

This small example introduces the following features of FaCiLe:

- The user interface to the library is provided by module `Facile` which gathers several specialized "sub-modules". We thus advise to open module `Facile` systematically to lighten FaCiLe functions calls. Most frequently used functions and sub-modules can then be directly accessed by opening module `Easy` (`open Easy`). Functions and modules names have been carefully chosen to avoid name clashes as much as possible with OCaml standard library when opening these two modules, but the "dot prefix" notation can still be used in case of fortuitous overlapping.

- The problem variables are created by a call to function `create` of module `Fd` (for **F**inite **d**omain, see 4.13) which takes a domain of type `Domain.t` as only argument. Domains are build and manipulated by functions of module `Domain` (see 4.4) like `Domain.create l` which creates a domain containing all integers of list `l`.
- `fd2e` and `i2e` constructs an expression respectively from a variable and an integer. More complex arithmetic expressions and constraints are build with infix operators (obtained by adding the suffix `~` to their integer counterparts) taking two expressions as arguments. Most usual arithmetic operators (not necessarily infix) are provided in module `Arith` (see 4.2).
- Function `post` from module `Cstr` adds a constraint to the constraint "store", which means that the constraint is taken into account and domain reduction is performed (as well as propagation on other variables).
- The search goal is here a simple labeling of the list of all the problem variables `[l;c;r;m]` obtained by a call to function `labeling` of sub-module `GlList` embedded in module `Goals` (see 4.7). The goal is therafter solved by a call to `solve` which returns `false` if a failure occured and `true` otherwise.
- The solution is then printed using function `fprint` from module `Fd`, which prints a variable on an output channel, i.e. its domain if the variable is not instantiated and its value otherwise.

This piece of code illustrates a typical FaCiLe CSP solving with the following pervasive ordered structure:

1. data and variables declaration
2. constraints statements
3. search goal specification
4. goal solving, i.e. searching solution(s)

In the next section, a more sophisticated example will help to describe more precisely how these very features can be easily implemented with FaCiLe.

1.2 A Classic Example

We solve now the even more recurrent cryptarithmetic problem $SEND + MORE = MONEY$ (see figure 1.2) where each letter stands for a distinct digit with $M \neq 0$ and $S \neq 0$.

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

Figure 1.2: The $SEND + MORE = MONEY$ problem

We model this problem with one variable for each digit plus three auxilliary variables to carry over, and the subsequent four arithmetic constraints specifying the result of the addition as we would do by hand. The following program implement this model:

smm.ml

```
open Facile
open Easy
let _ =
  (* Variables *)
```

```
  let s = Fd.interval 0 9 and e = Fd.interval 0 9 and n = Fd.interval 0 9
  and d = Fd.interval 0 9 and m = Fd.interval 0 9 and o = Fd.interval 0 9
  and r = Fd.interval 0 9 and y = Fd.interval 0 9 in
  (* Constraints *)
  Cstr.post (fd2e m >~ i2e 0);
  Cstr.post (fd2e s >~ i2e 0);
  let digits = [|s;e;n;d;m;o;r;y|] in
  Cstr.post (Alldiff.cstr digits);
  let c = Fd.array 3 0 1 in (* Carry array *)
  let one x = fd2e x and ten x = i2e 10 *~ fd2e x in
  Cstr.post (             one d +~ one e =~ one y +~ ten c.(0));
  Cstr.post (one c.(0) +~ one n +~ one r =~ one e +~ ten c.(1));
  Cstr.post (one c.(1) +~ one e +~ one o =~ one n +~ ten c.(2));
  Cstr.post (one c.(2) +~ one s +~ one m =~ one o +~ ten m);
  (* Search goal solving *)
  if Goals.solve (Goals.GlArray.labeling digits) then begin
    let value = Fd.min in
    Printf.printf "  %d%d%d%d\n" (value s) (value e) (value n) (value d);
    Printf.printf "+ %d%d%d%d\n" (value m) (value o) (value r) (value e);
    Printf.printf "=%d%d%d%d%d\n" (value m) (value o) (value n) (value e) (value
 y)
  end else
    prerr_endline "No solution"
```

```
unix% ocamlc facile.cma smm.ml
unix% ./a.out
  9567
+ 1085
=10652
```

We detail each step of the above example:

- Variables whose domain ranges an integer interval are created with function `Fd.interval inf sup` which creates a variable whose domain contains all integers between `inf` and `sup` (inclusive).
- Disequations $M \neq 0$ and $S \neq 0$ are then expressed by arithmetic inequality constraints and we assert that all digits must be distinct with the global `Alldiff.cstr` constraint which takes an array of variables as argument (see 4.1). FaCiLe provides some other global constraints as well, such as the global cardinality constraint (a.k.a. the "distribute" constraint) or the sorting constraint (see 4.6 and 4.11), usually embedded in separate module and called with function `cstr`.
- The three auxilliary carry variables are then created with `Fd.array n inf sup` which builds an array of `n` variables whose domains range the interval [`inf`..`sup`], and two auxilliary functions `one x` and `ten x` are defined which return an arithmetic expression being respectively equal to `x` and ten times `x` to lighten the main constraints statements.
- The equations reproducing the way we would compute the addition of figure 1.2 by hand are then straightforwardly stated an posted to the constraint store. The problem is finally solved as in the previous example by a simple labeling of the decision variables, i.e. the "digits", using function `labeling` of module `Goals.GlArray` (which is the counterpart of `GlList` over arrays of variables). The solution is then printed with function `Fd.min` which obviously returns the minimal value contained in the domain of a variable or its instantiation

value if the variable is ground (which is here the case as function `labeling` instantiates all the variables or raise a failure which would yield a `No solution` answer).

We could of course have used a different but equivalent model constraining the addition to be exact without auxilliary carry variables:

```
...
let op1 =
  i2e 1000 *~ fd2e s +~ i2e 100 *~ fd2e e +~ i2e 10 *~ fd2e n +~ fd2e d
and op2 =
  i2e 1000 *~ fd2e m +~ i2e 100 *~ fd2e o +~ i2e 10 *~ fd2e r +~ fd2e e in
let result =
  i2e 10000 *~ fd2e m +~
  i2e 1000 *~ fd2e o +~ i2e 100 *~ fd2e n +~ i2e 10 *~ fd2e e +~ fd2e y in
Cstr.post (op1 +~ op2 =~ op3);
...
```

This alternative model would undoubtedly produce the same result.

The next chapter will explore in a more formal way how to manipulate the main concepts of FaCiLe introduced in the two previous examples.

Chapter 2

Building Blocks

2.1 Domains

Finite domains of integers are created, accessed and manipulated with functions of module `Domain` (described exhaustively in section 4.4). They are represented as functional objects of (abstract) type `Domain.t` and can therefore be shared. Domains are build with different functions according to the domain property:

- `Domain.empty` is the empty domain;
- `Domain.create` is the most general constructor and builds a domain from a list of integers, possibly unsorted and with duplicates;
- `Domain.interval` is a shorthand when domains are continuous;
- `Domain.boolean` is a shorthand for `create [0;1]`;
- `Domain.int` is the largest (well, at least very large) domain.

Domains can be conveniently printed on an output channel with `Domain.fprint` and are displayed as lists of non-overlapping intervals and single integers `[`$\texttt{inf}_1$`-`$\texttt{sup}_1$`;`$\texttt{val}_2$`;`$\texttt{inf}_3$`-`$\texttt{sup}_3$`;...]` in increasing order:

```
#let discontinuous = Domain.create [4;7;2;4;-1;3];;
val discontinuous : Facile.Domain.t = <abstr>

#Domain.fprint stdout discontinuous;;
[-1;2-4;7]- : unit = ()

#let range = Domain.interval 4 12;;
val range : Facile.Domain.t = <abstr>

#Domain.fprint stdout range;;
[4-12]- : unit = ()
```

Various functions allow access to properties of domains like, among others (see 4.4), `Domain.is_empty`, `Domain.min`, `Domain.max` whose names are self-explanatory:

```
#Domain.is_empty range;;
- : bool = false

#Domain.max range;;
- : int = 12

#Domain.member 3 discontinuous;;
```

```
- : bool = true

#Domain.values range;;
- : int list = [4; 5; 6; 7; 8; 9; 10; 11; 12]
```

Operators are provided as well to manipulate domains and perform easily set operations like `Domain.intersection`, `Domain.union`, `Domain.difference` and domain reduction like `Domain.remove`, `Domain.remove_up`, `Domain.remove_low` etc (see 4.4):

```
#Domain.fprint stdout (Domain.intersection discontinuous range);;
[4;7]- : unit = ()

#Domain.fprint stdout (Domain.union discontinuous range);;
[-1;2-12]- : unit = ()

#Domain.fprint stdout (Domain.remove_up 3 discontinuous);;
[-1;2-3]- : unit = ()

#Domain.fprint stdout (Domain.remove_closed_inter 7 10 range);;
[4-6;11-12]- : unit = ()
```

2.2 Variables

FaCiLe variables are attributed objects[3] which maintain their current domain and can be backtracked during execution of search goals.

Creation

FaCiLe finite domain constrained variables are build and handled by functions of module `Var.Fd` (described exhaustively in section 4.13). Variables are objects of type `Fd.t` created by a call to one of the following functions of module `Var.Fd`:

- `create d` takes a domain `d` as argument.
- `interval inf sup` yields a variable whose domain ranges the interval `[inf..sup]`. It is equivalent to `create (Domain.interval inf sup)`.
- `array n inf sup` creates an array of `n` "interval" variables. Equivalent to `Array.init n ~f:(fun _ -> Fd.interval inf sup)`.
- `int n` returns a variable already bound to `n`.

The omnipresent `fprint` function writes a variable on an output channel:

```
#let vd = Fd.create discontinuous;;
val vd : Facile.Var.Fd.t = <abstr>

#Fd.fprint stdout vd;;
_4{[-1;2-4;7]}- : unit = ()
```

Attribute

A FaCiLe variable can be regarded as either in one of the two following states:

- *uninstantiated* or *unbound*, such that an "attribute" containing the current domain (of size strictly greater than one) is attached to the variable;
- *instantiated* or *bound*, such that merely an integer is attached to the variable.

So an unbound variable is associated with an attribute of type `Var.Attr.t` holding its current domain, a unique integer identifier and various data useless for the end-user. Functions to access attributes data are gathered in module `Var.Attr`:

- `dom` returns the current domain of an attribute;
- the mapping of `fprint`, `min`, `max`, `size`, `member` of module `Domain` applied on the embedded domain of an attribute (e.g. `min a` is equivalent to `Domain.min (dom a)`);
- `id` to get the identifier of an attribute;
- `constraints_number` returns the number of "active" constraints still attached to a variable attribute (see **??**).

Although variables are of abstract type `Fd.t`, function `Fd.value v` returns a concrete view of type `Var.concrete_fd = Unk of Attr.t | Val of int` [1] of a variable `v`, such that a control structure that depends on the instantiation of a variable will typically look like:

```
match Fd.value v with
    Val n -> f_bound n
  | Unk attr -> f_unbound attr
```

An alternative boolean function `Fd.is_var` returns the current state of a variable, sparing the "match" construct.

```
#let v1 = Fd.create (Domain.create [1]) (* equivalent to Fd.int 1 *);;
val v1 : Facile.Var.Fd.t = <abstr>

#Fd.is_var v1;;
- : bool = false

#Fd.fprint stdout v1;;
1- : unit = ()
```

Domain reduction

Module `Fd` provides three functions to perform backtrackable domain reductions on variables, typically used within instantiation goals and filtering of user-defined constraints:

- `subst v n` tries to instantiate variable `v` to integer `n`. `subst v n` fails whenever `n` does not belong to the domain of `v`. `v` must be unbound otherwise an exception is raised.

```
#let vr = Fd.interval 2 6;;
val vr : Facile.Var.Fd.t = <abstr>

#Fd.subst vr 7;;
Uncaught exception: Fcl_stak.Fail("Var.Fd.subst").

#Fd.subst vr 5;;
- : unit = ()

#Fd.fprint stdout vr;;
5- : unit = ()

#Fd.subst v1 1;;
Uncaught exception:
Failure
 "Fatal error: Fd.subst: bound variable (use Fd.unify on possible bound variable
)".
```

[1] Type `Var.concrete_fd` constructors `Unk` and `Val` stand respectively for "Unknown" (unbound) and "Value" (bound).

- `unify` is equivalent to `subst` but can be called on instantiated variables. If `v` is bound to `v_value`, then `unify v n` fails if `n` is different from `v_value`.

```
#Fd.unify v1 2;;
Uncaught exception: Fcl_stak.Fail("Var.Fd.unify").
```

```
#Fd.unify v1 1;;
- : unit = ()
```

- `refine v dom` reduces the domain of `v` to `dom`. `dom` **must be included in the current domain of** `v` otherwise an assert failure is raised with the byte code library `facile.cma` or the system will be corrupted with the optimized native code library `facile.cmxa`.

```
#Fd.fprint stdout vd;;
_4{[-1;2-4;7]}- : unit = ()
```

```
#match Fd.value vd with
#    Val n -> () (* Do nothing *)
#  | Unk attr -> (* Remove every value > 2 *)
#      let new_dom = Domain.remove_up 2 (Var.Attr.dom attr) in
#      Fd.refine vd new_dom;;
- : unit = ()
```

```
#Fd.fprint stdout vd;;
_4{[-1;2]}- : unit = ()
```

Whenever the domain of a variable becomes empty, a failure occurs (see **??** for more explanations about failure):

```
#match Fd.value vd with
#    Val n -> () (* Do nothing *)
#  | Unk attr -> (* Remove every value < 4 *)
#      let new_dom = Domain.remove_low 4 (Var.Attr.dom attr) in
#      Fd.refine vd new_dom;;
Uncaught exception: Fcl_stak.Fail("Var.Fd.refine").
```

Access

Besides `Fd.value` and `Fd.is_var` which access the state of a variable, module `Fd` provides the mapping of module `Domain` functions like `Fd.size`, `Fd.min`, `Fd.max`, `Fd.values`, `Fd.iter` and `Fd.member`, and they return meaningful values whatever the state (bound or unbound) of the variable may be:

```
#let vr = Fd.interval 5 8;;
val vr : Facile.Var.Fd.t = <abstr>
```

```
#Fd.size vr;;
- : int = 4
```

```
#let v12 = Fd.int 12;;
val v12 : Facile.Var.Fd.t = <abstr>
```

```
#Fd.member v12 12;;
- : bool = true
```

Function `Fd.id`, unlike the previous ones, returns a unique identifier for the variable only if it is uninstantiated, otherwise an exception is raised. An order based on these identifiers is defined by function `Fd.compare`[2] as well as an equality function `Fd.equal`, observing the two following rules:

[2]Comparison functions return 0 if both arguments are equal, a positive integer if the first is greater than the second and a negative one otherwise (like specified in the OCaml standard library).

1. bound variables are smaller than unbound variables
2. unbound variables are compared according to their identifiers

```
#Fd.id vr;;
- : int = 6

#Fd.id v12;;
Uncaught exception: Failure "Fatal error: Fd.id: bound variable".

#Fd.compare v12 (Fd.int 11);;
- : int = 1

#Fd.compare vr v12;;
- : int = 1

#Fd.id vd;;
- : int = 4

#Fd.compare vd vr;;
- : int = -1
```

2.3 Arithmetic expressions

Arithmetic expressions and constraints over finite domain variables are build with functions and operators of module `Arith` (see 4.2).

Creation and access

Arithmetic expressions are objects of abstract type `Arith.t` which contain a representation of an arithmetic term over finite domain variables. An expression is "*ground*" when all the variables used to build it are bound; in such a state an expression can be "evaluated" with function `Arith.eval` which returns its unique integral value. A call to `Arith.eval` with an expression that is not ground raises the exception `Invalid_argument`. However, any expression can be printed on an output channel with function `Arith.fprint`.

A variable of type `Fd.t` or an OCaml integer of type `int` **are not** arithmetic expressions and cannot therefore be mixed up with the latter. "Conversion" functions are provided by module `Arith` to build an expression from variables and integers :

- `Arith.i2e n` returns an expression which evaluates to integer `n`;
- `Arith.fd2e v` returns an expression which evaluates to `n` when `v` is instantiated and `Var.Fd.value v` evaluates to `Val n`.

Handily enough, opening module `Easy` allows direct access to most useful functions of module `Arith`, including `i2e` and `fd2e`:

```
#let v1 = Fd.interval 2 5;;
val v1 : Facile.Var.Fd.t = <abstr>

#let exp1 = fd2e v1;;
val exp1 : Facile.Arith.t = <abstr>

#Arith.fprint stdout exp1;;
_7{[2-5]}- : unit = ()

#Arith.eval exp1;;
Uncaught exception: Invalid_argument "Arith.eval: not ground".
```

```
#Fd.unify v1 4;;
- : unit = ()

#Arith.eval exp1;;
- : int = 4

#Arith.fprint stdout (i2e 2);;
2- : unit = ()
```

Maximal and minimal values of expressions can be accessed by functions `Arith.max_of_expr` and `Arith.min_of_expr`:

```
#let exp2 = fd2e (Fd.interval (-3) 12);;
val exp2 : Facile.Arith.t = <abstr>

#Arith.min_of_expr exp2;;
- : int = -3

#Arith.max_of_expr exp2;;
- : int = 12
```

An arithmetic expression can also be transformed into a variable thanks to function `Arith.e2fd` which creates a new variable constrained to be equal to its argument (see 2.4.2).

Operators

Module `Arith` provides classic linear and non-linear arithmetic operators to build complex expressions. They can be directly accessed through the opening of module `Easy`, which considerably ligthen the writting of equation, especially for binary infix ones.

- `+~`, `-~`, `*~`, `/~`: addition, substraction, multiplication and division (a failure occurs whenever its second argument evaluates to 0).
- `e **~ n` raises `e` to the `n`th power. `n` must be an integer.
- `x %~ y`: modulo, i.e. `x -~ (x /~ y) *~ y`. A failure occurs whenever `y` evaluates to 0.
- `Arith.abs`: absolute value.

```
#let vx = Fd.interval 3 6 and vy = Fd.interval 4 12;;


#let exp1 = i2e 2 *~ fd2e vx -~ fd2e vy +~ i2e 3;;
val exp1 : Facile.Arith.t = <abstr>

#Arith.fprint stdout exp1;;
(((2*_9{[3-6]})+-(_10{[4-12]}))+3)- : unit = ()

#Arith.min_of_expr exp1;;
- : int = -3

#Arith.max_of_expr exp1;;
- : int = 11
```

Global arithmetic operators working on array of expressions are provided as well:

- `Arith.sum exps` computes the sum of all the elements of the array of expressions `exps`.
- `Arith.scalprod ints exps` computes the scalar products of an array of integers by an array of expressions. `Arith.scalprod` raises `Invalid_argument` if the two arrays have not the same length.

- `Arith.prod exps` computes the product of all the elements of the array of expressions `exps`.

Their variable counterparts where the array of expressions is replaced by an array of variables are defined as well: `Arith.sum_fd`, `Arith.scalprod_fd`, `Arith.prod_fd`.

```
#let size = 5;;
val size : int = 5

#let coefs = Array.init size ~f:(fun i -> i+1);;
val coefs : int array = [|1; 2; 3; 4; 5|]

#let vars = Fd.array size 0 9;;
val vars : Facile.Var.Fd.t array =
  [|<abstr>; <abstr>; <abstr>; <abstr>; <abstr>|]

#let pscal_exp = Arith.scalprod_fd coefs vars;;
val pscal_exp : Facile.Arith.t = <abstr>

#Arith.fprint stdout pscal_exp;;
(((((0+(_11{[0-9]}*1))+(_12{[0-9]}*2))+(_13{[0-9]}*3))+(_14{[0-9]}*4))+(_15{[0-9
]}*5))- : unit = ()

#Arith.min_of_expr pscal_exp;;
- : int = 0

#Arith.max_of_expr pscal_exp;;
- : int = 135
```

2.4 Constraints

2.4.1 Creation and Use

A constraint in FaCiLe is a value of type `Cstr.t`. It can be created by a built-in function (arithmetic, global constraints) or user-defined (see **??**). A constraint must be *posted* with the function `Cstr.post` to be taken into account, i.e. added to the constraints store.

When a constraint is posted, it is attached to the involved variables and activated: propagation occurs as soon as the constraint is posted. The constraint is also stored in a global state accessible by the `Cstr.active_store` function which returns the list of all constraints still "unsolved", i.e. not yet globally consistent.

Constraints basically perform domain reductions on their involved variables, first when posted and then each time that a particular "event" occurs on their variables. An event corresponds to a domain reduction on a variable: the minimal or maximal value has changed, the size of the domain has decreased or the variable has been bound. To all these kinds of reduction are associated different events that will trigger the "awakening" of the appropriate constraints. See **??** for a more precise description of this event-driven mechanism.

Constraints can also be printed on an output channel with function `Cstr.fprint` which usually yields useful information about the variables involved and/or the name of the constraint.

2.4.2 Arithmetic Constraints

Simplest and standard constraints are relations on arithmetic expressions (c.f. 2.3):

- equality `=~`
- strict and non-strict inequality `<~`, `>~`, `<=~`, `>=~`
- disequality `<>~`

FaCiLe provides them as infix operators suffixed with the ~ character, similarly to expression operators. These operators are declared in the `Easy` module and don't need module prefix notation whenever `Easy` is opened. The small example below uses the equality operator `=~` and points out the effect on the variables domains of posting the constraint `equation`:

```
#(* 0<=x<=10, 0<=y<=10, 0<=z<=10 *)
#let x = Fd.interval 0 10 and y = Fd.interval 0 10 and z = Fd.interval 0 10;;

#let equation = (* x*y - 2*z >= 90 *)
#fd2e x *~ fd2e y -~ i2e 2 *~ fd2e z >=~ i2e 90;;
val equation : Facile.Cstr.t = <abstr>

#(* before propagation has occured *)
#Cstr.fprint stdout equation;;
 +2._18{[0-10]} -1._19{[0-100]} <= -90- : unit = ()

#Cstr.post equation;;
- : unit = ()

#(* after propagation has occured *)
#Cstr.fprint stdout equation;;
 +2._18{[0-5]} -1._19{[90-100]} <= -90- : unit = ()
```

FaCiLe compiles and simplifies ("normalizes") arithmetic constraints as much as possible such that variables and integers may be scattered inside an expression with no loss of efficiency. Therefore the constraint `ineq1`:

```
#let x = Fd.interval (-2) 6 and y = Fd.interval 4 12;;

#let xe = fd2e x and ye = fd2e y;;


#let ineq1 = i2e 3 *~ ye +~ i2e 2 *~ xe *~ ye *~ i2e 5 *~ xe +~ ye >=~ i2e 4300;;
val ineq1 : Facile.Cstr.t = <abstr>

#Cstr.fprint stdout ineq1;;
 -10._26{[0-432]} -4._23{[4-12]} <= -4300- : unit = ()
```

which ensures $3y + (2xy \times 5x) + y \geq 4300$, i.e. $10x^2y + 4y \geq 4300$, is equivalent to `ineq2`:

```
#let ineq2 = i2e 10 *~ (xe **~ 2) *~ ye +~ i2e 4 *~ ye >=~ i2e 4300;;
val ineq2 : Facile.Cstr.t = <abstr>

#Cstr.fprint stdout ineq2;;
 -10._31{[0-432]} -4._23{[4-12]} <= -4300- : unit = ()
```

Notice that the output of the `Cstr.fprint` function does not look exactly like the stated inequation but shows how the two operands of the main sum are internally reduced into new single variables constrained to be equal to the latters. This mechanism is of course hidden to the user and is only unfolded when calling `Cstr.fprint`.

Once posted, `ineq1` or `ineq2` incidentally yield a single solution:

```
#Printf.printf "x=%a y=%a\n" Fd.fprint x Fd.fprint y;;
x=_22{[-2-6]} y=_23{[4-12]}
- : unit = ()

#Cstr.post ineq1;;
- : unit = ()

#Printf.printf "x=%a y=%a\n" Fd.fprint x Fd.fprint y;;
x=6 y=12
- : unit = ()
```

It is also worthy to mention that arithmetic constraints involving (large enough) sums of boolean variables are automatically detected by FaCiLe and handled internally by a specific efficient mechanism. The user may thus be willing to benefit from these features by choosing a suitable problem modeling.

Note on precision and overflow

Users should be carefull when expecting the arithmetic solver to compute bounds from variables with very large domain, that means with values close to `max_int` or `min_int` (depending of the system and architecture). Especially with exponentiation and multiplication, an integer overflow may occur which will yield an exception if compiled in byte code and a wrong calculation if compiled in native code. An unexpected result when performing such operations in native code should thus always be checked against the byte code version.

Another possible source of miscalculation is the rounding performed when computing bounds of expressions involving exponentiation and multiplication or division: float operations are possibly performed with a loss of accuracy which is most of the time corrected by rounding, but errors might still occur for large numbers for which nth root or division requires a high precision (let's say 10^{-6}) not provided by available float operators. Please send a bug report if you think such a case occured (see in the *foreword*).

2.4.3 Global Constraints

Beside arithmetic constraints, FaCiLe provides so-called "global constraints" which express a relation on a set of variables. They are defined in separate modules in which a function (and possibly several variants) usually named `cstr` yields the constraint; these functions takes an array of variables as their main argument.

The most famous one is probably the "all different" constraint which expresses that all the elements of an array of variables must take different values. This constraint is invoked by the function `Alldiff.cstr ?algo vars` where `vars` is an array of variables and `algo` an optional argument that controls the efficiency of the constraint (see 4.1):

- `Lazy` waits for the instantiation of a variable and then removes the chosen value from the domains of the other variables;
- `Bin_matching evt` uses a more sophisticated algorithm (namely "bin matching" [2]) which is called whenever the event `evt` (see **??**) occurs on one of the array variable.

```
#let vars = Fd.array 5 0 4;;
val vars : Facile.Var.Fd.t array =
  [|<abstr>; <abstr>; <abstr>; <abstr>; <abstr>|]

#let ct = Alldiff.cstr vars;;
val ct : Facile.Cstr.t = <abstr>

#Fd.fprint_array stdout vars;;
[|_35{[0-4]}; _36{[0-4]}; _37{[0-4]}; _38{[0-4]}; _39{[0-4]}|]- : unit = ()

#Cstr.post ct; Fd.unify vars.(0) 3;;
- : unit = ()

#Fd.fprint_array stdout vars;;
[|3; _36{[0-2;4]}; _37{[0-2;4]}; _38{[0-2;4]}; _39{[0-2;4]}|]- : unit = ()
```

Module `FdArray` provides the "element" constraint named `FdArray.get` which allows to index an array of variables by a variable, and the `min` (and `max`) constraint which returns a variable constrained to be equal to the variable that will instantiate to the minimal (respectively maximal) value among the variables of an array:

```
#let vars = [|Fd.interval 7 12; Fd.interval 2 5; Fd.interval 4 8|];;
val vars : Facile.Var.Fd.t array = [|<abstr>; <abstr>; <abstr>|]

#let index = Fd.interval (-10) 10;;
val index : Facile.Var.Fd.t = <abstr>

#let vars_index = FdArray.get vars index;;
val vars_index : Facile.Var.Fd.t = <abstr>

#Fd.fprint stdout index;;
_66{[0-2]}- : unit = ()

#Fd.fprint stdout vars_index;;
_67{[2-12]}- : unit = ()

#let mini = FdArray.min vars;;
val mini : Facile.Var.Fd.t = <abstr>

#Fd.fprint stdout mini;;
_69{[2-5]}- : unit = ()
```

`FdArray.get` and `FdArray.min` which produce a new variable (and thus hide an underlying constraint) have also their "constraint" counterpart `FdArray.get_cstr` and `FdArray.min_cstr` which take an extra variable as argument and return a constraint of type `Cstr.t` that must be posted to be effective: `FdArray.min_cstr vars mini` is therefore equivalent to the constraint

```
fd2e (FdArray.min vars) =~ fd2e mini,
```

and `FdArray.get_cstr vars index v` to

```
fd2e (FdArray.get vars index) =~ fd2e v.
```

More sophisticated global constraints are available as well as FaCiLe built-in constraints:

- the global cardinality constraint [5] (a.k.a. "distribute" constraint): `Gcc.cstr` (see 4.6);
- the sorting constraint [1]: `Sorting.cstr` (see 4.11).

2.4.4 Reification

FaCiLe constraints can be "reified" thanks to the `Reify` module and its function `Reify.boolean` (see 4.10) which takes an argument of type `Cstr.t` and returns a new boolean variable. This boolean variable is interpreted as the truth value of the relation expressed by the constraint and the following equivalences hold:

- the boolean variable is bound to 1 iff the constraint is satisfied, and the constraint is thereafter posted;
- the boolean variable is bound to 0 iff the constraint is violated, and the negation of the constraint is thereafter posted;

otherwise, i.e. it is not yet known if the constraint is satisfied or violated and the boolean variable is not instantiated, the reification of a constraint do not perform any domain reduction on the variables involved.

In the following example, the boolean variable `is_x_less_than_y` is constrained to the truth value of the inequation constraint $x < y$:

```
#let x = Fd.interval 3 6 and y = Fd.interval 5 8;;
val x : Facile.Var.Fd.t = <abstr>
val y : Facile.Var.Fd.t = <abstr>

#let is_x_less_than_y = Reify.boolean (fd2e x <~ fd2e y);;
```

```
val is_x_less_than_y : Facile.Var.Fd.t = <abstr>

#Fd.fprint stdout is_x_less_than_y;;
_74{[0-1]}- : unit = ()

#Cstr.post (fd2e y >=~ i2e 7);;
- : unit = ()

#Fd.fprint stdout is_x_less_than_y;;
1- : unit = ()

#Fd.fprint stdout (Reify.boolean (fd2e x =~ fd2e y));;
0- : unit = ()
```

When posted, the reification of a constraint calls the `check` function (see **??**) of the constraint, which verifies whether it is satisfied or violated (without performing domain reduction). If it is violated, the negation of the constraint is posted with a call to another function of the constraint dedicated to reification, namely `not` (see **??**). Both functions are always defined for all constraints but their default behaviour is merely exception raising (`Failure "Fatal error: ..."`) which means that the constraint is actually not reifiable - as specified in the documentation of the relevant constraints in the reference manual. Mainly arithmetic constraints are reifiable (as well as the "interval" constraint of module `Interval`, see 4.8) and others (global ones) are not.

Reified constraint are by default waked up with the events triggering its standard awakening (i.e. when directly posted, not reified) *and* those of its negation. This behaviour might possibly be too time costly (for some specific problem) and the call to `Reify.boolean` with its optional argument `delay_on_negation` (see 4.10) set to `false` disables it, i.e. the events associated with the negation of the constraint are ignored.

Module `Reify` also provides standard logical (infix) operators over constraints:

- `&&~~`, conjunction;
- `||~~`, disjunction;
- `=>~~`, implication;
- `<=>~~`, equivalence;
- `not`[3], negation;

These operators can be directly accessed through the opening of module `Easy` except `Reify.not` (for obvious reasons). They can be combined to yield new logical operators like the "exclusive or" for example:

```
#let xor ct1 ct2 = Reify.not (ct1 <=>~~ ct2) in
#let x = Fd.interval 3 5 and y = Fd.interval 5 7 in
#let xor_cstr = xor (fd2e x =~ i2e 5) (fd2e y =~ i2e 5) in
#Cstr.post (xor_cstr);
#Cstr.post (fd2e x <=~ i2e 4);
#Fd.fprint stdout x;;
_82{[3-4]}- : unit = ()

#Fd.fprint stdout y;;
_72{[7-8]}- : unit = ()
```

Furthermore, module `Arith` contains convenient shortcuts to reify its basic arithmetic constraints:

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

[3]This one is obviously not infix.

These operators stand for the reification (and transformation into arithmetic expression) of their basic counterparts, i.e. they take two arithmetic expressions as operands and yield a new arithmetic expression being the boolean variable related to the truth value of the arithmetic constraint. `e1 =~~ e2` is therefore equivalent to

```
fd2e (Reify.boolean (e1 =~ e2))
```

These operators can also be directly accessed through the opening of module `Easy`. In the following example, the constraint stating that at least two of the three variables contained in array `vs` must be greater than 5 is expressed with the reified greater or equal `>=~~`:

```
#let vs = Fd.array 3 0 10;;
val vs : Facile.Var.Fd.t array = [|<abstr>; <abstr>; <abstr>|]

#Cstr.post (Arith.sum (Array.map ~f:(fun v -> fd2e v >~~ i2e 5) vs) >=~ i2e 2);
#Fd.fprint_array stdout vs;;
[|_91{[0-10]}; _92{[0-10]}; _93{[0-10]}|]- : unit = ()
```

If `vs.(1)` is forced to be less than 5, the two other variables become greater than 5:

```
#Cstr.post (fd2e vs.(1) <=~ i2e 5);
#Fd.fprint_array stdout vs;;
[|_91{[6-10]}; _92{[0-5]}; _93{[6-10]}|]- : unit = ()
```

2.5 Solving

Most constraint models are not tight enough to yield directly a single solution, such that search (and/or optimization) is necessary to find appropriate ones. FaCiLe uses *goals* to search for solutions. All built-in goals and functions to create and combine goals are gathered in module `Goals` (see 4.7). This section only introduces "ready-to-use" goals intended to implement basic search strategies, but more experimented users shall refer to sections **??** and **??** where combining goals with iterators, and building of goals from scratch are explained.

FaCiLe's most standard labeling goals is `Goals.indomain` which instantiates non-deterministically a single variable by disjunctively trying each value still in its domain in increasing order. To be executed, a goal must then be passed as argument of function `Goals.solve` which returns `true` if the goal succeeds, and `false` if it fails.

```
#let x = Fd.create (Domain.create [-4;2;12]);;
val x : Facile.Var.Fd.t = <abstr>

#Goals.solve (Goals.indomain x);;
- : bool = true

#Fd.fprint stdout x;;
-4- : unit = ()
```

So the first attempt to instantiate `x` (to -4) obviously succeeds.

The values of the domain of `x` can be enumerated with a slightly more sophisticated goal which fails just after `Goals.indomain`. Module `Goals` provides `Goals.fails`, which is a goal that fails immediately, and conjunction and disjunction operators, respectively `&&~` and `||~` (which can be directly accessed when module `Easy` is open), to combine simple goals. Hence such an enumeration goal would look like:

```
Goals.indomain x &&~ Goals.fail
```

But the result of such a goal will be failure and the state of the system (variable `x` not instantiated) will not be restored. A simple disjunction of this goal with the goal that succeeds immediately, `Goals.success`, yields the desirable behaviour :

```
Goals.indomain x &&~ Goals.fail ||~ Goals.success
```

In order to display the execution of this goal, a printing goal `gprint_fd` which prints a variable on the standard output (but will not be detailed in this section, see 3.4.1) can eventually be inserted (conjunctively) between `indomain` and `fail`:

```
#let x = Fd.create (Domain.create [-4;2;12]);;
val x : Facile.Var.Fd.t = <abstr>

#let goal = Goals.indomain x &&~ gprint_fd x &&~ Goals.fail ||~ Goals.success;;
val goal : Facile.Goals.t = <abstr>

#Goals.solve goal;;
-4 2 12 - : bool = true
```

Note that the logical operators have standard priorities.

Module `Goals` provides as well the function `Goals.instantiate` that allows to specify the odering strategy of the labeling. `Goals.instantiate` takes as first argument a function which is given the current domain of the variable (as single argument) and should return an integer candidate for instantiation. Labeling of variable `x` in decreasing order is then merely:

```
#let label_and_print labeling v =
#  labeling v &&~ gprint_fd v &&~ Goals.fail ||~ Goals.success;;
val label_and_print :
  (Facile.Var.Fd.t -> Facile.Goals.t) -> Facile.Var.Fd.t -> Facile.Goals.t =
  <fun>

#Goals.solve (label_and_print (Goals.instantiate Domain.max) x);;
12 2 -4 - : bool = true
```

Function `label_and_print` is defined here to lighten the writting of enumeration goals (it takes only the instantiation goal and the variable as arguments). In the example below, variable `x` is labeled in increasing order of the absolute value of its values. Function `Domain.choose` allows to specify only the relevant order:

```
#let goal =
#  label_and_print
#    (Goals.instantiate (Domain.choose (fun v1 v2 -> abs v1 < abs v2))) x;;
val goal : Facile.Goals.t = <abstr>

#Goals.solve goal;;
2 -4 12 - : bool = true
```

Beside non-deterministic instantiation, FaCiLe provides also `Goals.unify` to enforce the instantiation of a variable (which might be already bound) to a given integer value:

```
#Goals.solve (Goals.unify x 2);;
- : bool = true

#Fd.fprint stdout x;;
2- : unit = ()

#Goals.solve (Goals.unify x 12);;
- : bool = false

#Goals.solve (Goals.unify (Fd.int 0) 0);;
- : bool = true
```

2.6 Optimization

Classic Branch & Bound search is provided by the module `Opti` with the function `minimize`. It allows to solve a specified goal (`g`) while minimizing a cost defined by a finite domain variable (`c`):

1. Goal `g` is solved and its associated cost c (minimum of variable `c`) is computed;
2. a new constraint stating `c` $< c$ is added;
3. the process loops until `goal` fails.

The third argument of `Opti.minimize` is a function `f` called each time a solution is found. The argument of the function is the current cost (minimum of variable `c`).

`Opti.minimize` returns the result of the last call to function `f`. Because the function `f` may never be called, this result is packed with an `option` type. So if `goal` does not have any solution, returned value is `None`. If `goal` does have solutions and the search is not interrupted, the returned value is `(Some x)` where `x` is the value returned by function `f` when the last solution was found.

The following example solves the minimization of $x^2 + y^2$ while $x + y = 10$:

```
#let x = Fd.interval 0 10 and y = Fd.interval 0 10 in
#Cstr.post (fd2e x +~ fd2e y =~ i2e 10);
#let c = Arith.e2fd (fd2e x **~ 2 +~ fd2e y **~ 2) in
#let optimum =
#  Opti.minimize (Goals.indomain x &&~ Goals.indomain y) c
#    (fun c' -> Printf.printf "Found %d\n" c'; (Fd.min x, Fd.min y)) in
#match optimum with
#    None -> Printf.printf "No solution found\n"
#  | Some (x, y) ->
#      Printf.printf "Optimal solution: cost=%d x=%d y=%d\n" (Fd.min c) x y;;
Found 100
Found 82
Found 68
Found 58
Found 52
Found 50
Optimal solution: cost=50 x=5 y=5
- : unit = ()
```

Chapter 3

Advanced

3.1 Search Control

FaCiLe implements a standard depth first search with backtrack. OR control is handled with a stack (module `Stak`), while AND control is handled with continuations.

OR control can be modified with a cut à la Prolog: a level is associated to each choice-point (node in the search tree) and choice-points created since a specified level can be removed, i.e. *cut* (functions `Stak.level` and `Stak.cut`).

OR and AND controls are implemented by the `Goals.solve` function. AND is mapped on the imperative sequence. OR is based on the exception mechanism: backtrack is caused by the exception `Stak.fail` which is raised by failing constraints. Note that this exception is catched and handled only by the `Goals.solve` function.

3.1.1 Combining Goals with iterators

Functional programming allows the programmer to compose higher-order functions using *iterators*. An iterator is associated to a datatype and is the default control structure to process a value in the datatype. There is a strong isomorphism between the datatypes and the corresponding iterators and this isomorphism is a simple guideline to use them.

Imitating the iterators of the standard OCaml library, FaCiLe provides iterators for arrays and lists. While standard `Array` and `List` modules allows to construct sequences (with a `;`) of imperative functions (type `'a -> unit`), `GlArray` and `GlList` modules of FaCiLe allows to construct conjunction (with a `&&~`) and disjunction (with a `||~`) of goals (type `Goals.t`).

Arrays: module `Goals.GlArray`

Standard loop The polymorphic `Goals.GlArray.iter` function applies uniformally a goal to every element of an array, connecting them with a conjunction (`&&~`).

```
Goals.GlArray.iter g [|e1; e2; ...; en|] = (g e1) &&~ (g e2) &&~ ... &&~ (g en)
```

Labeling of an array of variables is the iteration of the instanciation of one variable (`Goals.indomain`):

```
#let labeling_array = Goals.GlArray.iter Goals.indomain;;
val labeling_array : Facile.Var.Fd.t array -> Facile.Goals.t = <fun>
```

A matrix is an array of arrays; following the isomorphism, labeling of a matrix must be simply a composition of the array iterator:

```
#let labeling_matrix = Goals.GlArray.iter labeling_array;;
val labeling_matrix : Facile.Var.Fd.t array array -> Facile.Goals.t = <fun>
```

Changing the order The `iter_h` (iterator with Heuristic) gives the user the possibility to choose the order the elements are considered. The first argument of `iter_h` is a function which is applied to the array by the iterator and which must return the index of one element on which the goal is applied. This function must raise the `Not_found` exception to stop the loop

For example, if we want to apply the goal only on unbound variables of an array, we may write:

```
#let first_unbound array =
#  let n = Array.length array in
#  let rec loop i = (* loop until free variable found *)
#    if i < n then
#      match Fd.value array.(i) with
#        Unk _ -> i
#      | Val _ -> loop (i+1)
#    else
#      raise Not_found in
#  loop 0;;
val first_unbound : Facile.Easy.Fd.t array -> int = <fun>

#let iter_unbounds = Goals.GlArray.iter_h first_unbound;;
val iter_unbounds :
  (Facile.Easy.Fd.t -> Facile.Goals.t) ->
  Facile.Easy.Fd.t array -> Facile.Goals.t = <fun>
```

Note that the function `iter_h` is polymorphic and can be used for any array.

The function `Goals.GlArray.choose_index` facilitates the construction of heuristic functions; it constructs such a function from an ordering function on variable attributes (free variables are ignored). For example, the standard "min size" strategy will be implemented as follows:

```
#let min_size_order =
#  Goals.GlArray.choose_index (fun a1 a2 -> Var.Attr.size a1 < Var.Attr.size a2);;
val min_size_order : Facile.Var.Fd.t array -> int = <fun>

#let min_size_strategy = Goals.GlArray.iter_h min_size_order;;
val min_size_strategy :
  (Facile.Var.Fd.t -> Facile.Goals.t) ->
  Facile.Var.Fd.t array -> Facile.Goals.t = <fun>

#let min_size_labeling = min_size_strategy Goals.indomain;;
val min_size_labeling : Facile.Var.Fd.t array -> Facile.Goals.t = <fun>
```

Lists: module `Goals.GlList`

FaCiLe `Goals.GlList` module provides similar iterators for lists except of course iterators which involve index of elements.

3.2 Constraints Control

Constraints may be seen operationnally as "reactive objects". They are attached to variables, more precisely to events related to variable modifications. A constraint is mainly a function (the *update* field) which is called when the constraint is *woken*. The update function usually do a propagation using the event (i.e. the modification of one variable) to process new domains for other variables.

3.2.1 Events

An event (of type `Var.Attr.event`) is a modification of the attribute (i.e. the domain) of a finite domain variable. FaCiLe currently provides only four specific events:

- Modification of the domain (`on_refine`);
- Substitution of the variable, i.e. reduction of the domain to a singleton (`on_subst`);
- Modification of the minimum value of the domain (`on_min`);
- Modification of the maximum value of the domain (`on_max`).

Note that these events are not independant and constitute a lattice which top is `on_subst` and bottom is `on_refine`:

- `on_subst` implies all other events[1];
- `on_min` and `on_max` imply `on_refine`.

Constraints are attached to the variables through these events. In concrete terms, lists of constraints (one per event) are put in the attribute of the variable. Note that this attachement occurs only when the constraint is posted.

3.2.2 Wakening, Queuing, Priorities

When an event occurs, related constraints are *woken* and put in a queue. The queue is processed after each sequence of waking. This processing is protected against reentrance. Constraints are considered one after the other and propagation (update) is called. Propagation may fail by raising an exception or succeed. Propagation of one constraint is also protected against rewaking by itself. When a constraint is triggered, the propagation does not know anything about the event and even does not get information about the variable responsible of the event. A constraint may be woken once by two distinct events. Note also that the queue contains constraints and not variables.

FaCiLe (currently) implements three ordered queues and ensures that a constraint in a lower queue is not propagated before a constraint present in a higher queue. The queue is chosen according to the *priority* of a constraint (abstract type `Cstr.priority`). The priority is specified when the constraint is defined. It cannot be changed neither when the constraint is posted nor later. Priorities are defined in module `Cstr`: `immediate`, `normal` or `later`.

3.2.3 Constraints Store

FaCiLe handles the constraints store of all the *posted* and *active* constraints (a constraint becomes inactive if it is solved, i.e. if the update returns true, see 3.3). For debugging purpose, this store can be consulted using the function `Cstr.active_store` and `Cstr.t` access functions (`id` and `name`).

3.3 User's Constraints

The `Cstr.create` function allows the user to build new constraints from scratch.

To define a new simple (unreifiable) constraint, very few arguments must be passed to the `create` function as numbers of them are optional (thus labeled) and have default values. Merely the two following arguments are actually needed to build a new constraint:

- `update` should perform propagation (domains reduction) and return true iff the constraint is consistent;

[1] It means that `on_min` event occurs if a variable is instantiated to its maximum value. This choice is arguable and could be thrown back into question into further releases.

- `delay` specifies on which events the `update` function will be called.

However we recommend to name a new constraint and precise its printing facility, which may obviously help debugging, by specifying the two following optional arguments:

- `?name` should be a relevant string describing the purpose of the constraint;
- `?fprint` to print more accurate information on the constraint state (variables domains, maintained data structures value, ...).

To define a reifiable constraint, the two following optional arguments must also be specified:

- `?check` should return true if the constraint is entailed, false if its negation is entailed and raise `DontKnow` otherwise. `check` is called when the constraint is reified and should therefore not perform any domain modification.
- `?not` should return the negation of the constraint. It is only called when the negation of a reified constraint is entailed.

Finally two other optional arguments may be specified:

- `?priority` should be passed to the `create` function to precise the priority of the new constraint in the constraints queue. Constraints with lower priority are waken only when there is no more constraint of higher priority in the waking queue. Time costly constraints should get a `later` while quick elementary constraints should be `immediate`, and standard constraints `normal` (default value).
- `?init` is executed as soon as the `post` function is called on the constraint to perform initialization of inner data structures needed by `update` (thus not called when dealing with a reified constraint).

The example below defines a new constraint stating that variable `x` should be different from variable `y`:

diff.ml

```
open Facile
open Easy

let cstr x y =
  let name = "different" in
  let fprint c =
    Printf.fprintf c "%s: %a <> %a\n" name Fd.fprint x Fd.fprint y
  and delay ct =
    Var.delay [Var.Attr.on_subst] x ct;
    Var.delay [Var.Attr.on_subst] y ct
  and update () =
    (* Domain reduction is performed only when x or y is instantiated *)
    match (Fd.value x, Fd.value y) with
      (Val a, Val b) -> a <> b || Stak.fail name
      (* If one of the two variables is instantiated, its value is
         removed in the domain of the other variable *)
    | (Val a, Unk attr_y) ->
        let new_domy = Domain.remove a (Var.Attr.dom attr_y) in
        Fd.refine y new_domy;
        true (* Constraint is solved *)
    | (Unk attr_x, Val b) ->
        let new_domx = Domain.remove b (Var.Attr.dom attr_x) in
```

```
        Fd.refine x new_domx;
        true (* Constraint is solved *)
    | _ -> false  (* Constraint is not solved *)
  and check () =
    match (Fd.value x, Fd.value y) with
      (Val a, Val b) -> a = b
    | (Val a, Unk attr_y) when not (Var.Attr.member attr_y a) -> true
    | (Unk attr_x, Val b) when not (Var.Attr.member attr_x b) -> true
    | _ -> raise Cstr.DontKnow
  and not () =
    fd2e x =~ fd2e y in
    (* Creation of the constraint. *)
  Cstr.create ~name ~fprint ~check ~not update delay;;
```

Let's compile the file

```
unix% ocamlc -c diff.ml
```

and use the produced object:

```
##load "diff.cmo";;

#let x = Fd.interval 1 2 and y = Fd.interval 2 3;;
val x : Facile.Easy.Fd.t = <abstr>
val y : Facile.Easy.Fd.t = <abstr>

#let diseq = Diff.cstr x y;;
val diseq : Facile.Cstr.t = <abstr>

#Cstr.post diseq;;
- : unit = ()

#let goal =
#  Goals.indomain x &&~ Goals.indomain y
#  &&~ Goals.create (fun () -> Cstr.fprint stdout diseq)
#  &&~ Goals.fail in
#while (Goals.solve  goal) do () done;;
different: 1 <> 2
different: 1 <> 3
different: 2 <> 3
- : unit = ()


#Cstr.one;;
- : Facile.Cstr.t = <abstr>

#Cstr.zero;;
- : Facile.Cstr.t = <abstr>
```

3.4 User's Goal

3.4.1 Atomic Goal: `Goals.create`

The simplest way to create a deterministic atomic goal is to use the `Goals.create` function which "goalify" any unit (i.e. of type `unit -> unit`) function.

Let's write the goal which writes a variable on the standard output:

```
#let gprint_fd x =
#  Goals.create (fun () -> Fd.fprint stdout x; print_newline ());;
val gprint_fd : Facile.Easy.Fd.t -> Facile.Goals.t = <fun>
```

To instantiate a variable inside a goal, we write the following definition :

```
#let instantiate x v = Goals.create (fun () -> Fd.unify x v);;
val instantiate : Facile.Easy.Fd.t -> int -> Facile.Goals.t = <fun>

#let v = Fd.interval 0 3 in
#if Goals.solve (instantiate v 2) then
#  Fd.fprint stdout v;;
2- : unit = ()
```

This goal creation can be used to pack any side effect function :

```
#let gprint_int x = Goals.create (fun () -> print_int x);;
val gprint_int : int -> Facile.Goals.t = <fun>

#Goals.solve (Goals.forto 0 5 gprint_int);;
012345- : bool = true
```

The main point when creating goals is to precisely distinguish the time of *creation* of the goal fromt the time of its *execution*. For example, the following goal does not produce what you maybe expect:

```
#let wrong_min_or_max var =
#  let min = Fd.min var and max = Fd.max var in
#  (instantiate var min ||~ instantiate var max);;
val wrong_min_or_max : Facile.Easy.Fd.t -> Facile.Goals.t = <fun>
```

The `min` and `max` of variable the `var` are processed when the goal is created and may be different from the min and max of the variable when the goal will be called. To fix the problem, `min` and `max` must be computed in the goal. Then the latter must return the disjunction, something it is not possible to do with `Goals.create`.

3.4.2 Arbitrary Goal: `Goals.make`

The `Goals.create` function does not allow to construct goals which construct new goals (similar to Prolog clauses). The `Goals.make` function "goalify" a function which may return another goal; the argument of `Goals.make` returns a `t option`, i.e. either `Some` (nothing) or `Some g` (the goal `g`).

Let's write the goal which try to instantiate a variable to its minimum value or to its maximum :

```
#let min_or_max v =
#  Goals.make
#    (fun () ->
#      let min = Fd.min v and max = Fd.max v in
#      Some (instantiate v min ||~ instantiate v max))
#    ();;
val min_or_max : Facile.Easy.Fd.t -> Facile.Goals.t = <fun>
```

The other difference of `Goals.make` with `Goals.create` is the argument of the goalified function which may be of any type (`'a`) and which must be passed as the second argument to `Goals.make`. In the previous example, we use `()`.

The `Goals.make` allows the user to define recursive goals by a mapping on a recursive function. In the next example, we iterate a goal non-deterministally on a list.

```
#let iter_disj fgoal list =
#  let rec loop l =
#    match  l with
#    | []    -> Some Goals.fail
#    | x::xs -> Some (fgoal x ||~ Goals.make loop xs) in
#  Goals.make loop list;;
val iter_disj : ('a -> Facile.Goals.t) -> 'a list -> Facile.Goals.t = <fun>

#let gprint_int x = Goals.create (fun () -> print_int x);;
val gprint_int : int -> Facile.Goals.t = <fun>

#let gprint_list = iter_disj gprint_int;;
val gprint_list : int list -> Facile.Goals.t = <fun>

#if Goals.solve (gprint_list [1;7;2;9] &&~ Goals.fail ||~ Goals.success) then
#  print_newline ();;
1729
- : unit = ()
```

3.4.3 Recursive Goals: `Goals.make_rec`

FaCiLe provides also a constructor for intrinsic recursive goals. Expression `[Goals.make_rec f]` is similar to `[Goals.make_rec f]` except that the argument of the function `f` is the created goal itself.

The simplest example using this feature is the classic `repeat` predicate of Prolog implementing a non-deterministic loop.

```
#let repeat = Goals.make_rec (fun self -> Some (Goals.success ||~ self));;
val repeat : Facile.Goals.t = <abstr>
```

The goalified function simply returned the disjunction of a success and itself.

The `Goals.indomain` function which non-non-deterministically instantiates a variable is written using `Goals.make_rec` :

```
#let indomain var =
#  Goals.make_rec
#    ~name:"indomain"
#    (fun self ->
#      match Fd.value var with
#        Val _ -> None
#      | Unk var_ ->
#         let dom = Var.Attr.dom var_ in
#         let remove_min =
#           Goals.create
#             (fun () -> Fd.refine var (Domain.remove_min dom))
#         and min = Domain.min dom in
#        Some (instantiate var min ||~ remove_min  &&~ self));;
val indomain : Facile.Easy.Fd.t -> Facile.Goals.t = <fun>
```

The goal first checks if the variable is already bound and do nothing in this case. If it is an unknown, it returns a goal trying to instantiate the variable to its minimum or to remove it before continuing with the remaining domain.

Part II

Reference Manual

Chapter 4

Modules

4.1 Module `Alldiff` : the "all different" constraint

```
type algo = Lazy | Bin_matching of Var.Attr.event
```

```
val cstr : ?algo:algo -> Var.Fd.t array -> Cstr.t
```

> `alldiff (?algo:Lazy) vars` Variables of `vars` are different from each other. `Lazy`: wait for instantiation and remove from other domains. `Bin_matching c`: wait for event `c` (e.g. `Var.Attr.on_refine`) and uses a bin matching algorithm to ensure the constraint is consistent. `algo` default value is `Lazy`. Not reifiable.

4.2 Module `Arith` : arithmetic expressions over variables of type `Var.Fd.t`

Overview

This module provides functions and operators to build arithmetic expressions and state (in—dis)equation constraints over them.

Basic

```
type t
```

> Type of arithmetic expressions over variables of type `Var.Fd.t` and integers.

```
val fprint : out_channel -> t -> unit
```

> `fprint chan e` print expression `e` on channel `chan`.

```
val eval : t -> int
```

> `eval e` return the integer numerical value of a fully instantiated expression `e`. Raise `Invalid_argument` if `e` is not instantiated.

```
val min_of_expr : t -> int
```

`min_of_expr e` return the minimal possible value of expression `e`.

```
val max_of_expr : t -> int
```

`max_of_expr e` return the maximal possible value of expression `e`.

Conversions

```
val i2e : int -> t
```

`i2e n` return an expression which evaluates to `n`.

```
val fd2e : Var.Fd.t -> t
```

`fd2e v` return an expression which evaluates to `n` when `v` is instantiated and `Var.Fd.value v` evaluates to `Val n`.

```
val e2fd : t -> Var.Fd.t
```

`e2fd e` create and return a new variable `v` and post the constraint `fd2e v =~ e`.

Construction of arithmetic expressions

The arithmetic operators check wether any integer overflow (i.e. the result of an arithmetic operation over integers is less than `min_int` or greater than `max_int`) occurs during constraints intern computations and raise an assert failure only when using the byte code library `facile.cma`.

```
val (+~) : t -> t -> t
```

Addition over expressions.

```
val ( *~) : t -> t -> t
```

Multiplication over expressions.

```
val (-~) : t -> t -> t
```

Substraction over expressions.

```
val (/~) : t -> t -> t
```

Division over expressions. A failure occurs whenever the second argument evaluates to 0.

```
val ( **~) : t -> int -> t
```

Exponentiation of an expression to an integers value.

```
val (%~) : t -> t -> t
```

Modulo over expressions.

```
val abs : t -> t
```

Absolute value over expressions.

```
val sum : t array -> t
```

Sum of all the elements of an array of expressions.

```
val sum_fd : Var.Fd.t array -> t
```

Sum of all the elements of an array of variables.

```
val scalprod : int array -> t array -> t
```

Scalar product of an array of integers and an array of expressions. Raise `Invalid_argument` if the arrays have not the same length.

```
val scalprod_fd : int array -> Var.Fd.t array -> t
```

Scalar product of an array of integers and an array of variables. Raise `Invalid_argument` if the arrays have not the same length.

```
val prod : t array -> t
```

Product of all the elements of an array of expressions.

```
val prod_fd : Var.Fd.t array -> t
```

Product of all the elements of an array of variables.

Arithmetic constraints over expressions

```
val (<=~) : t -> t -> Cstr.t
```

`e1 <=~ e2` return the constraint ensuring that the expression `e1` is less or equal than expression `e2`.

```
val (<~)  : t -> t -> Cstr.t
```

`e1 <~ e2` return the constraint ensuring that the expression `e1` is strictly less than expression `e2`.

```
val (>~)  : t -> t -> Cstr.t
```

`e1 >~ e2` return the constraint ensuring that the expression `e1` is strictly greater than expression `e2`.

```
val (=~)  : t -> t -> Cstr.t
```

`e1 =~ e2` return the constraint ensuring that the expression `e1` is equal to expression `e2`.

```
val (<>~)  : t -> t -> Cstr.t
```

`e1 =~ e2` return the constraint ensuring that the expression `e1` is different from expression `e2`.

```
val (>=~) : t -> t -> Cstr.t
```

`e1 >=~ e2` return the constraint ensuring that the expression `e1` is greater or equal than expression `e2`.

```
val even : t -> Cstr.t
```

`even e` return the constraint ensuring that that expression `e` is even.

```
val odd : t -> Cstr.t
```

`odd e` return the constraint ensuring that that expression `e` is odd.

Reification

Reify the arithmetic constraint which is replaced by an expression equal to a boolean variable instantiated to `1` if the constraint is satisfied and to `0` if it is violated.

```
val (<=~~) : t -> t -> t
```

Reified less or equal. `e1 <=~~ e2` is equivalent to `fd2e (Reify.boolean (e1 <=~ e2))`.

```
val (<~~)  : t -> t -> t
```

Reified strictly less.

```
val (>~~)  : t -> t -> t
```

Reified strictly greater.

```
val (>=~~) : t -> t -> t
```

Reified greater or equal.

```
val (=~~)  : t -> t -> t
```

Reified equality.

```
val (<>~~)  : t -> t -> t
```

Reified different.

4.3 Module `Cstr` : posting constraints and building new ones

Overview

This module defines the type `t` of constraints and functions to create and post constraints: mainly a `create` function which allows to build new constraints from scratch (this function is not needed when using standard FaCiLe predefined constraints) and the mostly useful `post` function which must be called to effectively add a constraint to the constraints store.

Basic

```
exception DontKnow
```

Exception raised by the `check` function of a constraint (of type `t`) when it is not known wether the constraint is satisfied or violated.

```
type priority
```

Waking priority.

```
val immediate : priority
```

As soon as possible, for quick updates.

```
val normal : priority
```

Standard priority.

```
val later : priority
```

For time costly constraints.

```
type t
```

The type of constraints

```
val id : t -> int
```

`id c` return a unique integer identifying the constraint `c`.

```
val name : t -> string
```

`name c` return the name of the constraint `c`.

```
val fprint : out_channel -> t -> unit
```

`fprint chan c` print the constraint `ct` on channel `chan`. Call the `fprint` function passed to `create`.

```
val create : ?name:string -> ?fprint:(out_channel -> unit) -> ?priority:prio
rity -> ?init:(unit -> unit) -> ?check:(unit -> bool) -> ?not:(unit -> t) ->
 (unit -> bool) -> (t -> unit) -> t
```

`create ?name ?fprint ?priority ?init ?check ?not update delay` build a new constraint.

- `name` is a describing string name of the constraint. Default value is `"anonymous"`.

- `fprint` (pretty-)printing of the constraint on an output channel taken as its only argument. Default value is to print the `name` string.

- `priority` is either `immediate`, `normal` or `later`. Time costly constraints should be waken after quick ones. Default value is `normal`.

- `init` is useful to perform initialization of auxiliary data structures needed and maintained by the `update` function. `init ()` is called as soon as the constraint is posted. Default value is `()`.

- `check` must be specified if the constraint is reifiable (as well as `not`). When the constraint is reified, `check ()` is called to verify whether the constraint is satisfied or violated, i.e. the constraint itself or its negation is entailed by the constraint store, and should return `true` if the constraint is satisfied, `false` if it is violated and raise `DontKnow` when it is not known. `check` must not change the domains of the variables involved in the constraint. Default : `Failure` exception raised.

- `not` must be specified if the constraint is reifiable (as well as `check`). `not ()` should return a constraint which is the negation of the constraint being defined. When the constraint is reified, it is called to post the negation of the constraint whenever `check ()` return `false`, i.e. it is entailed by the constraint store. Default : `Failure` exception raised.

- `update` compute the domain reductions of the constraint. `update ()` should return `true` when the constraint become solved, `false` if it is not yet entailed by the constraint store and raise `Stak.Fail` whenever a failure occurs. `update` is a mandatory argument.

- `delay` schedule the awakening of the constraint `ct` (which is taken as its only argument), i.e. the execution of its `update` function. If `update ()` should be called (because it may reduce variables domains) when one of the events contained in the events list `es` occured on variable `v`, then `Var.delay es v ct` should be called within the body of the `delay` function. `delay` is a mandatory argument.

`val post : t -> unit`

`post ct` post the constraint `ct` to the constraint store.

`val one : t`

The constraint which succeeds immediately.

`val zero : t`

The constraint which fails immediately.

`val active_store : unit -> t list`

`active_store ()` return the list of all active constraints, i.e. whose `update` functions have returned `false`.

4.4 Module `Domain` : domain operations

This module provides all necessary functions (and more) to create and manipulate domains, which are needed to create variables and perform propagation (i.e. domain reduction or filtering).

`type t`

Type of finite domains of integers (functional : no in place modifications, domains can be shared). Standard equality and comparison can be used on type domain.

Building new domains

`val empty : t`

The empty domain.

`val create : int list -> t`

`create l` build a new domain containing the values of `l`. Removes duplicates (and order is irrelevant). Return `empty` if `l` is empty.

`val interval : int -> int -> t`

`interval inf sup` return the domain of all integer in the closed interval `inf..sup`. Raise `Invalid_argument` if `inf > sup`.

`val int : t`

The largest representable domain.

`val boolean : t`

The domain containing `0` and `1`.

Access functions

`val is_empty : t -> bool`

Test if a domain is empty.

`val size : t -> int`

`size d` return the number of integers in `d`.

`val min : t -> int`

`min d` return the lower bound of `d`. If `d` is empty, the behaviour is unspecified (incorrect return value or exception raised).

`val max : t -> int`

`max d` return the upper bound of `d`. If `d` is empty, the behaviour is unspecified (incorrect return value or exception raised).

`val iter : (int -> unit) -> t -> unit`

`iter f d` applies `f` to every element of `d`

`val member : int -> t -> bool`

`member n d` test if `n` belongs to `d`.

`val values : t -> int list`

Return the list of values of a domain

`val fprint : out_channel -> t -> unit`

`fprint chan d` print `d` on channel `chan`.

`val sprint : t -> string`

`sprint d` returns a string representation of `d`.

`val included : t -> t -> bool`

`included d1 d2` test if domain `d1` is included in domain `d2`.

`val first_geq_value : t -> int -> int`

`first_geq_value dom val` return the smallest value in `dom` greater than `val`. Raise `Not_found` if max `dom < val`.

`val first_leq_value : t -> int -> int`

`first_leq_value dom val` return the greatest value in `dom` smaller than `val`. Raise `Not_found` if min `dom > val`.

`val choose : (int -> int -> bool) -> t -> int`

`choose ord d` returns the mininum value of `d` for order `ord`. Raise `Not_found` if `d` is empty

Operations

`val remove : int -> t -> t`

`remove n d` return d-n.

`val add : int -> t -> t`

`add n d` return `d`+`n`.

`val remove_up : int -> t -> t`

`remove_up n d` return `d`-`n`+1..`max_int`, i.e. removes values stricly greater than `n`.

`val remove_low : int -> t -> t`

`remove_low n d` return `d`-`min_int`..`n`-1, i.e. removes values stricly less than `n`.

`val remove_closed_inter : int -> int -> t -> t`

`remove_closed_inter inf sup d` return `d`-`inf`..`sup`, i.e. removes values greater or equal to `inf` and less or equal to `sup` in `d`. Return `d` if `inf > sup`.

`val intersection : t -> t -> t`

Intersection over domains.

`val union : t -> t -> t`

Union over domains.

`val difference : t -> t -> t`

`difference big small` return `big`-`small`. `small` must be included in `big`, otherwise the behaviour is unspecified (incorrect return value or exception raised).

`val remove_min : t -> t`

Remove the lower bound of a domain.

`val minus : t -> t`

Return the domain of opposite values.

`val plus : t -> int -> t`

`plus d n` translates a domain by `n`.

4.5 Module FdArray : constraints over arrays of variables

```
val min : Var.Fd.t array -> Var.Fd.t
```

min_array vars return a variable constrained to be equal to the variable that will be instantiated to the minimal value among all the variables in the array vars. Not reifiable.

```
val min_cstr : Var.Fd.t array -> Var.Fd.t -> Cstr.t
```

min_cstr vars mini return the constraint fd2e (min vars) =~ fd2e mini. Not reifiable.

```
val max : Var.Fd.t array -> Var.Fd.t
```

Idem as min but for the maximal value.

```
val max_cstr : Var.Fd.t array -> Var.Fd.t -> Cstr.t
```

Idem as min_cstr but for the maximal value.

```
val get : Var.Fd.t array ->  Var.Fd.t -> Var.Fd.t
```

get vars index return a variable constrained to be equal to vars.(index). Not reifiable.

```
val get_cstr : Var.Fd.t array -> Var.Fd.t -> Var.Fd.t -> Cstr.t
```

get_cstr vars index v return the constraint fd2e vars.(index) =~ fd2e v. Not reifiable.

4.6 Module Gcc : global cardinality constraint (aka distribute)

```
type level = Basic | Medium | High
```

```
val cstr : ?level:level -> Var.Fd.t array -> (Var.Fd.t * int) array -> Cstr.
t
```

cstr (?level:High) vars distribution return a constraint ensuring that for each pair (c,v) of cardinal variable c and integer value v in the list distribution, c variables in the array vars will be instantiated to v, i.e. card{vi = v | vi in vars} = c. Three levels of propagation are provided : Basic is the quickest, High performs the highest amount of propagation. level default value is High. Not reifiable.

4.7 Module Goals : building and solving goals

Overview

This module provides functions and operators to build goals that will control the search, i.e. mainly choose and instantiate variables.

Goals type, creation, ...

`type t`

The type of goals.

`val create : ?name:string -> (unit -> unit) -> t`

`create (?name:"create") f` return a goal executing `f`. `f` must take `()` as argument and return `()`. `name` default value is `"create"`.

`val make : ?name:string -> ('a -> t option) -> 'a -> t`

`make (?name:"make") f a` return a goal calling `f a`. `f` may return a goal (embedded in an option type). `name` default value is `"make"`.

`val make_rec : ?name:string -> (t -> t option) -> t`

`make_rec (?name:"make_rec") f` return a goal executing `f`. `f` takes itself as argument and may return a goal (embedded in an option type). Useful to write recursive goals. `name` default value is `"make_rec"`.

`val fail : t`

Failure. Neutral element for the disjunction over goals. Could be implemented as `create (fun () -> raise (Stak.fail "fail"))`.

`val success : t`

Success. Neutral element for the conjunction over goals. Could be implemented as `create (fun () -> ())`.

`val (&&~) : t -> t -> t`

Conjunction over goals.

`val (||~) : t -> t -> t`

Disjunction over goals.

`val once : t -> t`

`once g` cut choice points left on goal `g`.

`val solve : ?control:(int -> unit) -> t -> bool`

`solve ?control g` solve the goal `g` and return a success (`true`) or a failure (`false`). The execution can be controlled precisely with the `control` argument whose single argument is the number of bactracks since the beginning of the search. This function is called after every local failure:

- it can raise `Success` to stop the search and a success (`true`) is thereafter returned;

- it can raise `Stak.Fail` to force a failure of the search in the current branch (i.e. backtrack);

- it can raise any other user exception to stop the search process;

- it must return `unit` to continue the search; this is the default behavior.

```
val unify : Var.Fd.t -> int -> t
```

unify var x instantiates variable var to x

```
val indomain : Var.Fd.t -> t
```

Non-deterministic instanciation of a variable, by labeling its domain (in increasing order).

```
val instantiate : (Domain.t -> int) -> Var.Fd.t -> t
```

instantiate choose var Non-deterministic instanciation of a variable, by labeling its domain using the value returned by the choose function.

```
val dichotomic : Var.Fd.t -> t
```

Non-deterministic instanciation of a variable, by dichotomic recursive exploration of its domain.

```
val forto : int -> int -> (int -> t) -> t
```

forto min max g conjunctive iteration of goal g on increasing integers from min to max.

```
val fordownto : int -> int -> (int -> t) -> t
```

fordownto min max g conjunctive iteration of goal g on decreasing integers from max to min. Equivalent to forto min max (fun i -> g (max - i + min))

Goal operation on array of variables.

```
module GlArray :  sig
```

```
    val iter : ('a -> t) -> 'a array -> t
```

iter g a return the conjunctive iteration of goal g on array a. iter g [|x1;x2;...;xn|] is g x1 &&~ g x2 &&~ ... &&~ g xn

```
    val iteri : (int -> 'a -> t) -> 'a array -> t
```

iter g a return the conjunctive iteration of goal g on integer step of iteration i and array a. iteri g [|x1;x2;...;xn|] is g 0 x1 &&~ g 1 x2 &&~ ... &&~ g (n-1) xn

```
    val labeling : Var.Fd.t array -> t
```

Equivalent to iter indomain.

```
    val iter_h : ('a array -> int) -> ('a -> t) -> 'a array -> t
```

iter_aray_h h g a return the conjunctive iteration of goal g where order is computed by the heuristic h which must raise Not_found to terminate, i.e. at each iteration g is applied to a.(h a).

```
    val iter_hi : ('a array -> int) -> (int -> 'a -> t) -> 'a array -> t
```

iter_aray_hi h g a return the conjunctive iteration of goal g where order is computed by the heuristic h which must raise Not_found to terminate, i.e. at the ith iteration g is applied to a.(h i a).

```
val choose_index : (Var.Attr.t -> Var.Attr.t -> bool) -> Var.Fd.t array -> i
nt
```

`choose_index order fds` return the index of the best (minimun) free (not instantiated) variable in the array `fds` for the criterion `order`. Raise `Not_found` if all variables are bound (instantiated).

```
val not_instantiated_fd : Var.Fd.t array -> int
```

`not_instantiated_fd fds` return the index of one element in `fds` which is not instantiated. Raises `Not_found` if all variables in array `fds` are bound.

Goal operations on list of variables.

```
module GlList :  sig
```

```
val iter : ('a -> t) -> 'a list -> t
```

`iter g [x1;x2;...;xn]` is `g x1 &&~ g x2 &&~ ... &&~ g xn`, i.e. return the conjunctive iteration of goal `g` on list `a`.

```
val labeling : Var.Fd.t list -> t
```

Equivalent to `iter indomain`.

```
val iter_h : ('a list -> 'a*('a list)) -> ('a -> t) -> 'a list -> t
```

`iter_h h g l` return the conjunctive iteration of goal `g` where order is computed by the heuristic `h`, i.e. at each iteration `h l` returns the couple `(v,vs)` of the variable (or data) which will be passed as the argument of goal `g` (thus executing `g v`) and the list `vs` on which the iteration will go on.

4.8 Module `Interval` : variable membership of an interval

```
val is_member : Var.Fd.t -> int -> int -> Var.Fd.t
```

`is_member v inf sup` return the boolean variable instantiated to `1` if `v` is in `inf..sup` and to `0` otherwise.

```
val cstr : Var.Fd.t -> int -> int -> Var.Fd.t -> Cstr.t
```

`cstr v inf sup b` return the constraint ensuring that the boolean variable `b` is instantiated to `1` if `v` is in `inf..sup` and to `0` otherwise. Not reifiable.

4.9 Module `Opti` : Branch and Bound optimization

```
val minimize : Goals.t -> Var.Fd.t -> ?control:(int -> unit) -> ?step:int ->
 (int -> 'a) -> 'a option
```

`minimize goal cost ?control (?step:1) solution` runs a Branch and Bound algorithm on `goal` for bound `cost`, with an improvment of a least `step` between each solution found. `solution` is called with the instantiation value of `cost` as argument each time a solution is found. The result is the value returned by `solution` on the last solution (if any). `?control` is passed to `Goals.solve` as its first (optional) argument. Default `control` does nothing (i.e. `fun _ -> ()`). Default `step` is 1.

4.10 Module Reify : constraints reification

```
val boolean : ?delay_on_negation:bool -> Cstr.t -> Var.Fd.t
```

boolean (?delay_on_negation:true) c returns a boolean (0..1) variable associated to the constraint c. The constraint is verified iff the boolean is equal to 1. The waking conditions of the contraint relating c and the boolean are the ones set by the delay method of c (set by the delay argument of Cstr.create). If the optional argument delay_on_negation is set to true, the new constraint is also attached to the events of the negation of c (i.e. the constraint returned by the not method of c), otherwise it is not. delay_on_negation default value is true.

```
val cstr : ?delay_on_negation:bool -> Cstr.t -> Var.Fd.t -> Cstr.t
```

cstr delay_on_negation c b is equivalent to the constraint boolean ?delay_on_negation c =~ b. delay_on_negation default value is true.

```
val (&&~~) : Cstr.t -> Cstr.t -> Cstr.t
```

```
val (||~~) : Cstr.t -> Cstr.t -> Cstr.t
```

```
val (=>~~) : Cstr.t -> Cstr.t -> Cstr.t
```

```
val (<=>~~) : Cstr.t -> Cstr.t -> Cstr.t
```

```
val not : Cstr.t -> Cstr.t
```

Logical operations on constraints (which are reified), namely and, or, implies, equivalent, not.

4.11 Module Sorting : sorting constraint

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

sort a return an array of variables constrained to be the variables in a sorted in increasing order.

```
val sortp : Var.Fd.t array -> Var.Fd.t array * Var.Fd.t array
```

sortp a same as sort but return a couple (sorted, perm) where sorted is the array of sorted variables and perm is an array of variables constraint to be the permutation between a and sorted, i.e. a.(i) = sorted.(perm.(i)).

```
val cstr : Var.Fd.t array -> ?p:Var.Fd.t array option -> Var.Fd.t array -> C
str.t
```

cstr a (?perm:None) sorted return the constraint ensuring that sorted is the result of sorting array a according to the permutation perm. perm default value is None, meaning the argument is irrelevant. Not reifiable.

4.12 Module Stak : global stack of goals, backtrackable operations

This module provides functions to control the execution of the goal stack, as well as backtrackable references. Nota: the module name `Stak` lacks a 'c' because of a clash with the Ocaml standard library module `Stack`.

```
type level
```

Type of a level of the stack.

```
exception Level_not_found of level
```

Raised by `cut` and `cut_bottom`.

```
val size : unit -> int
```

Size of the stack.

```
val level : unit -> level
```

Return the current level.

```
val cut : level -> unit
```

`cut l` cut the choice points left on the stack until level `l`. Raise `Level_not_found` if level is not reached and stack is empty.

```
type 'a ref
```

Backtrackable reference of type `'a`

```
val ref : 'a -> 'a ref
```

Return a reference which modification will be backtracked during the solving of a goal.

```
val set : 'a ref -> 'a -> unit
```

Set a reference.

```
val get : 'a ref -> 'a
```

Dereference.

Control of failure

```
exception Fail of string
```

Raised during solving whenever a failure occurs. The string argument is informative.

```
val fail : string -> 'a
```

`fail x` is equivalent to `raise (Fail x)`.

4.13 Module Var : constrained, attributed, finite domain variables

The attribute of a domain variable

```
module Attr :  sig
```

`type t`

Type of attributes.

`val dom : t -> Domain.t`

`dom a` return the integer domain of an attribute.

`type event`

Type of events (modifications on variables) on which to suspend a constraint.

`val on_refine : event`

Event occuring when a variable is changed, i.e. its domain modified.

`val on_subst : event`

Event occuring when a variable is instantiated.

`val on_min : event`

Event occuring when the lower bound of a variable increases.

`val on_max : event`

Event occuring when the upper bound of a variable decreases.

`val fprint : out_channel -> t -> unit`

`fprint chan a` print attribute `a` on channel `chan`.

`val size : t -> int`

`size a` return the number of integer values in the domain associated with `a` (i.e. `dom a`).

`val min : t -> int`

`min a` return the lower bound of `a`.

`val max : t -> int`

`max a` return the upper bound of `a`.

`val values : t -> int list`

`values a` return the list of all integers in `dom a`.

`val iter : t -> (int -> unit) -> unit`

`iter a f` iter f on each integer in `dom a`.

`val member : t -> int -> bool`

`member a n` test if `n` belongs to `dom a`.

`val id : t -> int`

`id a` return a unique integer identifying the attribute `a`.

`val constraints_number : t -> int`

`constraints_number a` return the number of different constraints attached to `a`.

A domain variable

```
type concrete_fd = Unk of Attr.t | Val of int
```

Type of the value of a Fd variable.

```
module Fd :  sig
```

Finite domain variable

```
type t
```

Type of Fd variable.

```
val create : Domain.t -> t
```

`create d` return a new variable with domain `d`.

```
val interval : int -> int -> t
```

`interval inf sup` return a new variable with domain `inf..sup`.

```
val array : int -> int -> int -> t array
```

`array n inf sup` return an array of `n` new variables with domain `inf..sup`.

```
val int : int -> t
```

`int n` return a new variable instantiated to integer value `n`.

```
val is_var : t -> bool
```

`is_var v` test if `v` is not instantiated.

```
val value : t -> concrete_fd
```

`value v` return `Val n` if `v` is instantiated to `n`, `Unk a` otherwise where `a` is the attribute of `v`. Should always be used in a matching: `match value v with Val n -> ...  | Unk a -> ....`

```
val subst : t -> int -> unit
```

`subst v n` instantiate variable `v` with integer value `n`. Raise `Stak.Fail` in case of failure. Must be called only on unbound (not instantiated) variable, otherwise a `Failure` exception is raised.

```
val refine : t -> Domain.t -> unit
```

`refine v d` reduce the domain of `v` with domain `d`. `d` must be included in the domain of `v`, otherwise the behaviour is unspecified (corrupted system or exception raised). Must be called only on non ground variable (`Failure` exception otherwise).

```
val unify : t -> int -> unit
```

Idem as subst but can be called on instantiated variable.

```
val fprint : out_channel -> t -> unit
```

`fprint chan v` print variable `v` on channel `chan`.

```
val fprint_array : out_channel -> t array -> unit
```

`fprint_array chan vs` print array of variables `vs` on channel `chan`.

```
val size : t -> int
```

`size v` return the number of integer values in the domain of `v` (`1` if `v` is instantiated.

`val min : t -> int`

`min v` return the lower bound of `v`.

`val max : t -> int`

`max v` return the upper bound of `v`.

`val values : t -> int list`

`values v` return the list of all integers in the domain of `v`. If `v` is instantiated to `n`, return the singleton list containing `n`.

`val iter : t -> (int -> unit) -> unit`

`iter v f` iter f on each integer in the domain of `v`.

`val member : t -> int -> bool`

`member v n` test if `n` belongs to the domain of `v`.

`val id : t -> int`

`id v` return a unique integer identifying the attribute associated with `v`. Must be called only on non ground variable, raise Failure otherwise.

`val compare : t -> t -> int`

Compare two variables. Values (bound variables) are smaller than unknowns (unbound variables). Unknowns are sorted according to their attribute `id`.

`val equal : t -> t -> bool`

Test if two variables are equal with respect to `compare`.

`val delay : Attr.event list -> Fd.t -> Cstr.t -> unit`

`delay event_list v c` suspend constraint `c` on all the events in `event_list` occuring on `v`. No effect on instantiated variables.

4.14 Module Easy

`Easy` is a module the user must open to facilitates access to FaCiLe. It simply defines aliases to values and types of other modules:

- All the infix operators from `Arith` and `Goals`;
- Frequently used mapping functions of `Arith`: `i2e` and `fd2e`;
- Type of finite domain variables from `Var`: `concrete_fd`;
- Module `Fd` from `Var`

Note that the user of FaCiLe can extend this mechanism with its own `Easier` module aliasing any value or type of the library.

Index

Bibliography

[1] Noelle Bleuzen Guernalec and Alain Colmerauer. Narrowing a $2n$-block of sorting in $O(n \log n)$. In *Principles and Practice of Constraint Programming*. Springer-Verlag, 1997.

[2] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchig in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, 1973.

[3] S. Le Huitouze. A new data structure for implementing extensions to Prolog. In P. Deransart and J. Małuszyński, editors, *2nd Int. Work. Programming Languages Implementation and Logic Programming, LNCS 456*, pages 136–150. Springer-Verlag, 1990.

[4] Xavier Leroy. The Objective Caml System: User's and reference manual (`http://caml.inria.fr`), 2000.

[5] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.