FaCiLe: A Functional Constraint Library
Release 1.1

N. Barnier P. Brisset

September 8, 2004

Preface

FaCiLe is a constraint programming library over integer finite domains written in OCaml [7]. It offers all usual constraints system facilities to create and handle finite domain variables, arithmetic expressions and constraints (possibly non-linear), built-in global constraints and search goals. FaCiLe allows to easily build user-defined constraints and goals (including recursive ones) from scratch or by combining simple primitives, making pervasive use of OCaml higher-order functionals to provide a simple and flexible user interface. 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 these features of OCaml (among many others) allow to prototype and experiment quickly: modeling, data processing and interface are implemented with the same powerful and efficient language.

This manual is not a document about constraint programming techniques but only a tool description. Users should be familiar with other constraint systems to easily apprehend FaCiLe through the reading of this manual. Beginners can easily find comprehensive information on the Web (e.g. `http://www.cs.unh.edu/ccc/archive/`). This manual is neither a course about functional programming and users should refer to the official Caml Web site at `http://caml.inria.fr/` and the OCaml manual [7] to obtain introductory (as well as advanced) material about the host language of FaCiLe. Hurried readers may also take a look at a short overview appearing in the ALP Newsletter [2]. Thorough ones may find deeper insights on FaCiLe implementation details unveiled in the second part of one of the author PhD thesis [1].

Since OCaml forbids overloading, FaCiLe unusual looking operators might be a little disconcerting at first sight. Moreover, there is no implicit casting, so explicit conversions between variables (or integer) and arithmetic expressions are compulsory. These features lead to less concise expressions than with poorly typed languages, however the user precisely knows which operation is executed by the system and cannot erratically mix values of different types. Furthermore, ML style higher-order functionals and powerful type system ease the design and processing of complex data structures without the need of syntactic sugar (iterators, mapping and folding are "native" in OCaml). So FaCiLe does not endlessly provide more and more ad hoc functions for each particular case to exhibit the smallest possible code for toy examples, but rather aims at featuring simple building blocks and operators to combine them efficiently.

This manual is structured in two main parts:

1. The user's manual which starts with basic examples to give a taste of FaCiLe, then details the main concepts and eventually discusses more advanced subjects like the design of constraints and goals from scratch.

2. The reference manual which describes module by module all the functionalities of FaCiLe. This part of the documentation is automatically generated from the source code interface files (`.mli`), which may be directly consulted.

Numerous examples are provided all along the user's manual and more complete ones are available within the standard distribution in the `examples` directory, as well as a generic `Makefile` do build FaCiLe / OCaml softwares.

Eventually, we would like to thank our early known beta-testers, Mattias Waldau and Pal-Kristian Engstad, whose suggestions helped us to improve FaCiLe.

Good reading.

Foreword

Portability

FaCiLe requires only the OCaml system (release 3.02 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 split into numerous modules and submodules. They are all included (possibly with a limited user-oriented interface) 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`). The pseudo-module named `Easy` is the exception and should be opened: it provides several aliases to the most frequently used values (see 4.1) and functions.

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 lot of checks are done in this mode

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

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 -I +facile facile.cma
```

This is the toplevel used in the inlined examples of this documentation and invoked with the command line:

```
./facile -I +facile
```

Availability

The FaCiLe distribution and documentation are available from the web site 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. Below is a copy of the corresponding part:

```
INSTALLATION:

  All you need is the Objective Caml 3.02 (or greater) compiler and
standard Unix tools (make...).

  0) Configure the library. The single option of configuration is the
directory you want to put the library files in (facile.cma, facile.cmxa,
facile.a facile.cmi). Default is the subdirectory "facile" of the Ocaml
library directory (returned by "ocamlc -where").

./configure [--faciledir <target directory>]

  1) First compile the library with a simple

        make

  2) Check the result

        make check

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

```
3) Then install the library with a (usually as root)

      make install
```

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:

Coins Give back change for any amount

Golf Organize a golf tournament for 8 teams of 4 players

Golomb Find optimal Golomb rulers

Jobshop Solve the famous `mt10` scheduling problem – Edge-Finding inside!

Magic To count and to be counted

Marriage Stabilize preferences among spouses

Prolog Use FaCiLe as a Prolog interpreter on a family tree problem

Queens Place queens on a chessboard

Seven_eleven My grocer's favorite arithmetic puzzle

Tiles Tile a big square with small squares

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 example) must be compiled with the library of object byte code `facile.cma` when batch compiling with `ocamlc`:

```
ocamlc -I +facile facile.cma csp.ml
```

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

```
ocamlopt -I +facile facile.cmxa csp.ml
```

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

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

```
ocamlmktop -o facile -I +facile facile.cma
```

and invoked with:

```
./facile -I +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 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 has to repaint the Canadian flag (shown in figure 1.1) with its two original colors, red and white, so that two neighbouring areas don't have the same color and the maple leaf is... red, of course. 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 adjacency $l \neq c$, $c \neq r$, $m \neq c$ and the maple leaf is forced to be red $m = 0$.

The following piece of code solves this problem:

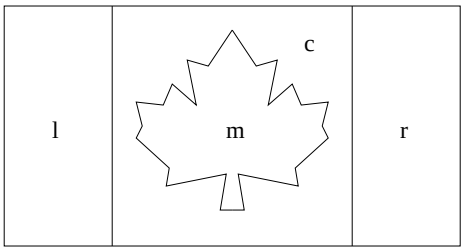

Figure 1.1: The problem of the Canadian flag

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.List.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 -I +facile facile.cma maple.ml
unix% ./a.out
l=0 c=1 r=0 m=0
```

Surprisingly enough, the new flag is 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 "submodules". We thus advise to open module `Facile` systematically to lighten FaCiLe functions calls. Most frequently used functions and submodules 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 a fortuitous overlapping.

- The problem variables are created by a call to function `create` of module `Fd` (for **F**inite **d**omain, see **??**) which takes a domain of type `Domain.t` as only argument. Domains are built and handled by functions of module `Domain` (see **??**) 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 built 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 **??**).
- 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).
- Here the search goal is a simple labeling of the list of all the problem variables `[l;c;r;m]` obtained by a call to function `labeling` of submodule `List` embedded in module `Goals` (see **??**). The goal is thereafter solved by a call to `solve` which returns `false` if a failure occurred 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 precisely describe how these 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.Array.labeling digits) then begin
    let value = Fd.elt_value 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 -I +facile facile.cma smm.ml
unix% ./a.out
  9567
+ 1085
=10652
```

We detail each step of the above example:

- Variables whose domains range integer intervals 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 **??**). 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 **??** and **??**), 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 and posted to the constraint store. The problem is finally solved as in the first example by a simple labeling of the decision variables, i.e. the "digits", using function `labeling` of module `Goals.Array` (which is the counterpart of `Goals.List` over arrays of variables). The solution is then printed with function `Fd.elt_value` which returns the integer value of an instantiated variable (or raises an exception whenever it is still unbound).

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 handle the main concepts of FaCiLe introduced in the two previous examples.

Chapter 2

Building Blocks

FaCiLe offers variables and constraints on integer and set finite domains. This chapter first describes how to build a constraint program on standard integer variables, while explaining the basics underlying concepts of FaCiLe. Then section 2.7 extends the scheme to set variables, which work in a similar fashion.

2.1 Domains

Finite domains of integers are created, accessed and handled with functions of module `Domain` (exhaustively described in section **??**). Domains basically are sets of "elements" of type `Domain.elt` (here integers, or sets of integers for the set domains described in section 2.7.1). They are represented as functional objects of (abstract) type `Domain.t` and can therefore be shared. Domains are built with different functions according to the domain properties:

- `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 $[\texttt{inf}_1\texttt{-sup}_1\texttt{;val}_2\texttt{;inf}_3\texttt{-sup}_3\texttt{;...}]$ 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 **??**), `Domain.is_empty`, `Domain.min`, `Domain.max` whose names are self-explanatory:

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

#Domain.max range;;
- : Facile.Domain.elt = 12

#Domain.member 3 discontinuous;;
- : bool = true

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

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

```
#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[6] which maintain their current domain and can be backtracked during the execution of search goals.

Creation

FaCiLe finite domain constrained variables are build and handled by functions of module `Var.Fd` (described exhaustively in section **??**). 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 (fun _ -> Fd.interval inf sup)`.
- `int n` returns a variable already bound to `n`.

Note that the submodule `Fd` can be reached by opening module `Easy`; in all the toplevel examples, modules `Facile` and `Easy` are supposed open, therefore a function `f` of module `Fd` is called with `Fd.f` instead of `Facile.Var.Fd.f`.

The first three creation functions actually have an optional argument labelled `?name` which allows to associate a string identifier to a variable. The ubiquitous `fprint` function writes a variable on an output channel and uses this string name if provided or an internal identifier if not:

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

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

Attribute

A FaCiLe variable can be regarded as either in one of the following two 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, its string name, a unique integer identifier and various internal data irrelevant to 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.

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`, so 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 two functions to perform backtrackable domain reductions on variables, typically used within instantiation goals and filtering of user-defined constraints:

- `unify v n` instantiates variable `v` to integer `n` or fails whenever `n` does not belong to the domain of `v`. `unify` may be called on instantiated variables.

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

#Fd.unify vr 7;;
Exception: Fcl_stak.Fail "Var.XxxFd.subst".

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

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

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

#Fd.unify v1 2;;
Exception: Fcl_stak.Fail "Var.XxxFd.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;;
vd[-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;;
vd[-1;2]- : unit = ()
```

Whenever the domain of a variable becomes empty, a failure occurs (see 2.5 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;;
Exception: Fcl_stak.Fail "Var.XxxFd.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
```

Contrarily, function `Fd.id`, which returns the unique identifier associated with a variable, or function `Fd.name`, which returns its specified string name, only work if the variable is still uninstantiated, otherwise an exception is raised.

An order based on the integer identifiers is defined by function `Fd.compare`[2] as well as an equality function `Fd.equal`, observing the following two rules:

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

```
#Fd.id vr;;
- : int = 2

#Fd.id v12;;
Exception: Failure "Fatal error: Var.XxxFd.id: bound variable".

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

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

#Fd.id vd;;
- : int = 0

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

Eventually, function `Fd.elt_value` returns the integer value of a bound variable. If the variable is not instantiated, an exception is raised.

```
#Fd.elt_value (Fd.int 1);;
- : Facile.Var.Fd.elt = 1

#Fd.elt_value (Fd.interval 0 1);;
Exception: Failure "Fatal error: Var.XxxFd.elt_value: unbound variable: _3".
```

2.3 Arithmetic Expressions

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

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 therefore cannot 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 bound to `n`.

[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 (as specified in the OCaml standard library).

Handily enough, opening module `Easy` allows direct access to most useful functions and operators 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;;
_4[2-5]- : unit = ()

#Arith.eval exp1;;
Exception: Failure "Fatal error: Expr.eval: variable _4 unknown".

#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
```

Conversely, an arithmetic expression can 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. Most frequently used ones can be directly accessed through the opening of module `Easy`, which considerably ligthen the writing of equations, especially for binary infix operators.

- `+~`, `-~`, `*~`, `/~`: addition, substraction, multiplication and division (the exception `Division_by_zero` is raised whenever its second argument evaluates to 0).
- `e **~ n` raises `e` to the `n`th power, where `n` is an integer.
- `x %~ y`: modulo. The exception `Division_by_zero` is raised whenever `y` evaluates to 0.
- `Arith.abs`: absolute value.

```
#let vx = Fd.interval ~name:"x" 3 6 and vy = Fd.interval ~name:"y" 4 12;;


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

#Arith.fprint stdout exp1;;
3 + -y[4-12] + 2 * x[3-6]- : 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` builds the sum of all the elements of the array of expressions `exps`.
- `Arith.scalprod ints exps` builds 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` builds 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`. Note that `Arith.sum_fd a`, for example, is simply defined as `Arith.sum (Array.map fd2e a)`.

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

#let coefs = Array.init size (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;;
1 * _8[0-9] + 2 * _9[0-9] + 3 * _10[0-9] + 4 * _11[0-9] + 5 * _12[0-9]- : 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 3.3). A constraint must be *posted* with the function `Cstr.post` to be taken into account, i.e. added to the *constraint store*. The state of the system can then be accessed by a call to the function `Cstr.active_store` which returns the list of all constraints still "unsolved", i.e. not yet globally consistent.

When a constraint is posted, it is attached to the involved variables and activated: propagation occurs as soon as the constraint is posted. Consequently, if an inconsistency is detected prior to the search, i.e. before the call to `Goals.solve` (see 2.5), a `Stak.Fail` exception is raised. However, inconsistencies generally occur during the search so that failures are caught by the goal solving mechanism of FaCiLe which will backtrack until the last choice-point.

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. All these kinds reduction cause different events to trigger the "awakening" of the appropriate constraints. See 3.2.1 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

The 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 occurred *)
#Cstr.fprint stdout equation;;
3:  +2._15[0-10] -1._16[0-100] <= -90- : unit = ()

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

#(* after propagation has occurred *)
#Cstr.fprint stdout equation;;
3:  +2._15[0-5] -1._16[90-100] <= -90- : unit = ()
```

Notice that the output of the `Cstr.fprint` function does not look exactly like the stated inequation but gives a hint about 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 using the pretty-printer.

FaCiLe compiles and simplifies ("normalizes") arithmetic constraints as much as possible so 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;;
6:  -4._18[4-12] -10._20[0-432] <= -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;;
9:  -4._18[4-12] -10._22[0-432] <= -4300- : unit = ()
```

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

```
#Printf.printf "x=%a y=%a\n" Fd.fprint x Fd.fprint y;;
x=_17[-2-6] y=_18[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 worth mentioning 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. This automatic behaviour can be tuned by specifying the minimum size from which the constraint is optimized (see **??**).

Note on Overflows

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 on the system and architecture). Especially with exponentiation and multiplication, an integer overflow may occur which will yield an error message ("Fatal error: integer overflow") on `stderr` and an exception (`Assert_failure`) **if the program is compiled in byte code**. A spurious calculation (probably leading to a failure during propagation) will happen if it is compiled in native code. An unexpected behaviour when performing such operations in native code should thus always be checked against the safer byte code version.

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 (of type `Alldiff.algo`) that controls the efficiency of the constraint (see **??**):

- `Lazy` waits for the instantiation of a variable and then removes the chosen value from the domains of the remaining variables;
- `Bin_matching evt` uses a more sophisticated algorithm (namely a "bin matching" [5]) which is called whenever the event `evt` (see 3.2.1) occurs on one of the variables to globally check the satisfiability of the constraint.

```
#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;;
[|_23[0-4]; _24[0-4]; _25[0-4]; _26[0-4]; _27[0-4]|]- : unit = ()

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

#Fd.fprint_array stdout vars;;
[|3; _24[0-2;4]; _25[0-2;4]; _26[0-2;4]; _27[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` (resp. `max`) constraint which returns a variable constrained to be equal to the variable that will instantiate to the minimal (resp. 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;;
_31[0-2]- : unit = ()

#Fd.fprint stdout vars_index;;
_32[2-12]- : unit = ()

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

#Fd.fprint stdout mini;;
_33[2-5]- : unit = ()
```

`FdArray.get` and `FdArray.min`, which produce a new variable (and thus hide an underlying constraint), also have 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 [9] (a.k.a. "distribute" constraint): `Gcc.cstr` (see **??**);
- the sorting constraint [3]: `Sorting.cstr` (see **??**).

2.4.4 Reification

FaCiLe constraints can be "reified" thanks to the `Reify` module and its function `Reify.boolean` (see **??**) 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 does not perform any domain reduction on the variables involved.

In the following example, the boolean variable `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 x_less_than_y = Reify.boolean (fd2e x <~ fd2e y);;
val x_less_than_y : Facile.Var.Fd.t = <abstr>

#Fd.fprint stdout x_less_than_y;;
_36[0-1]- : unit = ()

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

#Fd.fprint stdout 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 3.3) 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 3.3). 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. Roughly, arithmetic constraints are reifiable (as well as the "interval" constraint of module `Interval`, see **??**) while others (global ones) are not.

Reified constraint are by default woken up with the events triggering its standard awakening (i.e. as if it were directly posted) **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 **??**) set to `false` disables it, i.e. the events associated with the negation of the constraint are ignored.

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

- `&&~~`, conjunction;
- `||~~`, disjunction;
- `=>~~`, implication;
- `<=>~~`, equivalence;

- `xor`[3], exclusive or;
- `not`[3], negation.

These operators can be directly accessed through the opening of module `Easy`, except `Reify.not` (for obvious reasons) and `Reify.xor` (which are not infix). Note that, unlike `Reify.boolean`, these operators do not have a `?delay_on_negation` optional argument, so that the constraints they return will be woken by both the events of their arguments and those of the negations of their arguments.

These operators can be combined to yield complex logical operators. For example, the "exclusive or" may be redefined in the following way:

```
#let x = Fd.interval 3 5 and y = Fd.interval 5 7;;
val x : Facile.Var.Fd.t = <abstr>
val y : Facile.Var.Fd.t = <abstr>

#let xor ct1 ct2 = Reify.not (ct1 <=>~~ ct2) in
#let xor_cstr = xor (fd2e x =~ i2e 5) (fd2e y =~ i2e 5) in
#Cstr.post (xor_cstr);
#Cstr.post (fd2e x <=~ i2e 4);
#Printf.printf "x=%a y=%a\n" Fd.fprint x Fd.fprint y;;
x=_38[3-4] y=5
- : unit = ()
```

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

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

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 (fun v -> fd2e v >~~ i2e 5) vs) >=~ i2e 2);
#Fd.fprint_array stdout vs;;
[|_40[0-10]; _41[0-10]; _42[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;;
[|_40[6-10]; _41[0-5]; _42[6-10]|]- : unit = ()
```

[3]Not infix.

2.5 Search

Most constraint models are not tight enough to yield directly a single solution, so 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 **??**). This section only introduces "ready-to-use" goals intended to implement basic search strategies, but more experienced users shall refer to sections 3.1.2 and 3.4, where combining goals with iterators and building 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 to function `Goals.solve` which returns `true` if the goal succeeds or `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.fail`, which is a goal that always fails, 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 a 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 always succeeds, `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, unfortunately, **the logical operators do have the same priority**. Hence goals expressions must be carefully parenthesized to produce the expected result.

Module `Goals` also provides the function `Goals.instantiate` that allows to specify the ordering strategy of the labeling. `Goals.instantiate` takes as first argument a function to 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 writing of enumeration goals (it takes only the instantiation goal and the variable as arguments). In the example below, variable `x` is labelled in increasing order of the absolute value of its values. Function `Domain.choose` allows to only specify 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
```

Search strategy Like most CP system, FaCiLe default standard strategy is Depth First Search. However, FaCiLe now offers Limited Discrepancy Search [4] as well (see **??**), and even if a general mechanism to change the search strategy is not provided, skilled users are encouraged to plunder and hack the source code of module `Goals` to devise new custom strategies themselves.

Floundering If the search goal does not instantiate all the variables involved in the posted constraints, some of the constraints may still be unsolved when a solution is found, so that this solution may be incorrect. To be sure that all the constraints have been solved, the user can use the function `Cstr.active_store` and checks that the returned constraints list is empty. This checking may be done after the completion of the search, i.e. after `Goals.solve`, or better, embedded within the search goal. The latter allows to cleanly integrate this verification in optimization and "findall" goals. A "non-floundering check" goal could be implemented in the following way (function `Goals.atomic` used here to build a new atomic goal is explained in section 3.4.1):

```
#let check_floundering =
#  Goals.atomic
#    (fun () ->
#      if Cstr.active_store () <> [] then
#        failwith "Some constraints are still unsolved");;
val check_floundering : Facile.Goals.t = <abstr>
```

A simple conjunction with `check_floundering` at the end of the labeling goal will do the job. Information about the alive constraints may be extracted as well, thanks to module `Cstr` access functions (`id`, `name`, `fprint`).

Early Backtrack With FaCiLe as in Prolog systems, any dynamic modification performed within goals may be undone (backtracked) to restore the state of the system. However, no choice-point is associated to the "root" of the constraint program, so that variables modifications occurring before the call to `Goals.solve` can never be undone. As the standard way of adding constraints with FaCiLe is to post them prior to the solving, i.e. statically outside goals, the domain reductions initially made by these constraints are not backtrackable.

2.6 Optimization

Classic Branch & Bound search is provided by the function `minimize` of module `Goals`. 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 the cost must then be bound to a value *cc*, i.e. the current cost of the current solution

2. Backtracking is performed to restore the state of the system as before the execution of `g` and a new constraint stating `c` $< cc$ is added to the constraint store

3. The process loops until `goal` fails

The third argument of `Goals.minimize` is a function `solution : int -> unit` called each time a solution is found. The argument of `solution` is the current value of the cost `cc` which *must* be instantiated by `g`. This function is handy to store the last solution and cost in references, because `Goals.minimize` *always fails*, so that the decision and cost variables are restored as before its execution by `Goals.solve`.

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 store = ref None in
#let solution cc =
#  store := Some (cc, Fd.elt_value x, Fd.elt_value y);
#  Printf.printf "Found %d\n" cc in
#let g = Goals.minimize (Goals.indomain x &&~ Goals.indomain y) c solution in
#if Goals.solve (g ||~ Goals.success) then
#  match !store with
#    None -> Printf.printf "No solution\n"
#  | Some (best_c, best_x, best_y) ->
#      Printf.printf "Optimal solution: cost=%d x=%d y=%d\n" best_c best_x best_y;;
Found 100
Found 82
Found 68
Found 58
Found 52
Found 50
Optimal solution: cost=50 x=5 y=5
- : unit = ()
```

Additionally, `Goals.minimize` has two optional arguments:

- `?step`: the improvement between two consecutive solutions must be greater than `step`, i.e. the constraint posted each time a solution is found is `c` $\leq cc -$ `step`; `step` default value is obviously 1.

- `?mode`: may be either `Goals.Restart` or `Goals.Continue` (of type `bb_mode`); with mode `Restart`, the search restarts from the beginning at each step, i.e. the system backtracks until the very state prior to the execution of `minimize`, whereas with mode `Continue` the search simply carries on with an update of the cost constraint. Default mode is `Goals.Continue`.

2.7 Constraint Programs on Finite Sets

CP can be parameterized by the mathematical structure on which to express variables and constraints. In (almost) the same way, FaCiLe uses the generic mechanism of *functors* to provide variables either on integers domain or on finite sets (of integers) domain. Hence, the interface (of type `BASICFD`, see **??**) on which variables are built is the same for both types (and then further extended for integer ones), once parameterized by the `Domain` module, and once by the `SetDomain` one.

So the few previous sections are relevant to document set variables and constraints. Specific issues are discussed below.

2.7.1 Set Domains

The standard `Domain` module builds domain (of type `Domain.t`) from its basic elements, integers, whose type is aliased as `Domain.elt`. Similarly, the `SetDomain` module builds domain of type `SetDomain.t` from basic elements, set of integers with type `SetDomain.elt`. The latter type simply is an alias for type `SetDomain.S.t` of module `SetDomain.S` which provides values and functions to build and handle elements of `SetDomain` (see **??**).

Set domains represent sets of integers sets. They are described as powerset lattices of sets bounded by its definite elements, the *glb* (Greater Lower Bound) and possible elements *lub* (Lower Upper Bound). So the glb corresponds to the `min` value of an integer domain while the lub corresponds to its `max`.

Figure 2.1 illustrates the representation of the following domain :

```
#let glb = SetDomain.elt_of_list [1;2];;
val glb : Facile.SetDomain.elt = <abstr>

#let lub = SetDomain.elt_of_list [1;2;3;4;5];;
val lub : Facile.SetDomain.elt = <abstr>

#let sd = SetDomain.interval glb lub;;
val sd : Facile.SetDomain.t = <abstr>

#SetDomain.fprint stdout sd;;
{ 1 2 }..{ 1 2 3 4 5 }- : unit = ()
```

Note that the glb must be included in the lub, and that "holes" cannot be represented at the domain level.

2.7.2 Set Variables

The module defining set variables, `SetFd`, shares its interface with module of integer variables `Fd`:

```
#let sv = Var.SetFd.interval ~name:"sv" glb lub;;
val sv : Facile.Var.SetFd.t = <abstr>

#Var.SetFd.fprint stdout sv;;
sv{ 1 2 }..{ 1 2 3 4 5 }- : unit = ()

#Var.SetFd.unify sv (SetDomain.S.empty);;
Exception: Fcl_stak.Fail "Var.XxxFd.subst".
```

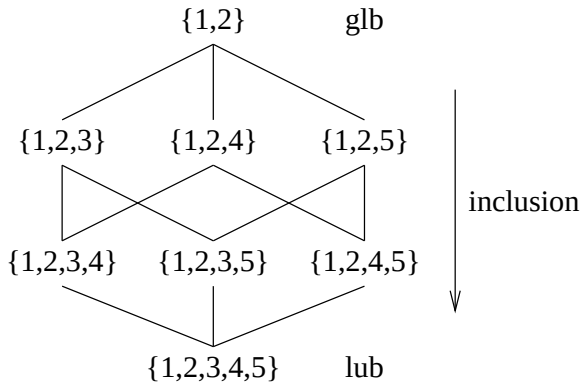

Figure 2.1: Lattice of a set domain

However, specific (convenient) set operations (and constraints) are located in module `Conjunto`:

```
#Conjunto.inside 5 sv;;
- : unit = ()

#Var.SetFd.fprint stdout sv;;
sv{ 1 2 5 }..{ 1 2 3 4 5 }- : unit = ()
```

2.7.3 Constraints

Constraints on set variables can be found in module `Conjunto` (see **??**). Set operators like union, intersection, subset... are provided, as well as operators involving integer variables like cardinality or membership. The following example defines a fixed set `super` and its 2-partition as sets `sub1` and `sub2`. It uses the union, disjoint and cardinal constraints of module `Conjunto`:

```
#let lub = SetDomain.elt_of_list [1;2];;
val lub : Facile.SetDomain.elt = <abstr>

#let super = Var.SetFd.interval lub lub;;
val super : Facile.Var.SetFd.t = <abstr>

#let sub1 = Var.SetFd.interval SetDomain.S.empty lub
#and sub2 = Var.SetFd.interval SetDomain.S.empty lub;;
val sub1 : Facile.Var.SetFd.t = <abstr>
val sub2 : Facile.Var.SetFd.t = <abstr>

#let card = Conjunto.cardinal (Conjunto.union sub1 sub2);;
val card : Facile.Var.Fd.t = <abstr>

#Cstr.post (Conjunto.disjoint sub1 sub2);;
- : unit = ()

#Cstr.post (fd2e card =~ i2e (SetDomain.S.cardinal lub));;
- : unit = ()
```

2.7.4 Labeling

A specific goal is provided within module `Goals.Conjunto` to non-deterministically instantiate set variables. The following example enumerates and prints the 2-partitions of set `super`:

```
#let print () =
#  Printf.printf "sub1=%a sub2=%a\n" Var.SetFd.fprint sub1 Var.SetFd.fprint sub2 in
#let g =
#  Goals.Conjunto.indomain sub1 &&~ Goals.Conjunto.indomain sub2
#  &&~ Goals.atomic print &&~ Goals.fail in
#ignore (Goals.solve g);;
sub1={ } sub2={ 1 2 }
sub1={ 2 } sub2={ 1 }
sub1={ 1 } sub2={ 2 }
sub1={ 1 2 } sub2={ }
- : unit = ()
```

Chapter 3

Advanced Usage

3.1 Search Control

3.1.1 Basic Mechanisms

FaCiLe implements a standard depth-first search with backtracking. 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 operationally 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 caught and handled by the `Goals.solve` function only.

3.1.2 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`), `Goals.Array` and `Goals.List` modules of FaCiLe allows to construct conjunction (with a `&&~`) and disjunction (with a `||~`) of goals (type `Goals.t`).

The simplest iterator operates on integers and provides a standard *for-to* loop by applying a goal to consecutive integers:

```
Goals.forto 3 7 g = (g 3) &&~ (g 4) &&~ ... &&~ (g 7)
```

Of course, iterators may be composed, as is illustrated below, where the cartesian product $[1..3] \times [4..5]$ is deterministically enumerated:

```
#let enum_couples =
#  Goals.forto 1 3
#    (fun i ->
#      Goals.forto 4 5
#        (fun j ->
#          Goals.atomic (fun () -> Printf.printf "%d-%d\n" i j))) in
#Goals.solve enum_couples;;
1-4
```

```
1-5
2-4
2-5
3-4
3-5
- : bool = true
```

Function `Goals.atomic` (used in the previous example), which builds an "atomic" goal (i.e. a goal which returns nothing), is detailed in section 3.4.1.

Arrays: module `Goals.Array`

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

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

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

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

Changing the Order An optional argument of `Goals.Array.forall`, labelled `?select`, gives the user the possibility to choose the order in which the elements are considered. `?select` 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 exception `Not_found` to stop the loop.

For example, if we want to apply the goal only on the 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 forall_unbounds = Goals.Array.forall ~select:first_unbound;;
val forall_unbounds :
  (Facile.Easy.Fd.t -> Facile.Goals.t) ->
  Facile.Easy.Fd.t array -> Facile.Goals.t = <fun>
```

Note that the function `forall` is polymorphic and can be used for an array of any type.

The function `Goals.Array.choose_index` facilitates the construction of heuristic functions that may be provided to the `forall ?select` argument. 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.Array.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.Array.forall ~select: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>
```

Note that module `Goals.Array` also provides a disjunctive iterator, `exists`, which has the same profile than `forall`. Variants `Goals.Array.foralli` and `Goals.Array.existsi` allow to specify goals which take the index of the relevant variable as an extra argument (like the OCaml standard library iterator `Array.iteri`).

Lists: module `Goals.List`

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

3.2 Constraints Control

Constraints may be seen operationally as "reactive objects". They are attached to variables, more precisely to *events* related to variable modifications. A constraint mainly is an *update* function (responsible for performing propagations) which is called when the constraint is *woken* because a specific event occured. Events are queued according to the *priority* of the constraint, and the search control is resumed as soon as all queues are emptied.

3.2.1 Events

An event (of type `Var.Fd.event`) is a modification of the domain of a variable. FaCiLe currently provides 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, thanks to the `Var.Fd.delay`[2] function. 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.

[1] It means that, e.g. the event `on_min` occurs even if a variable is instantiated to its minimum value.
[2] Or `Var.SetFd.delay` for set variables.

3.2.2 Suspending to Events, Waking Identity

Constraints are suspended to events by invoking the `delay` function wich takes an events list and an optional integer *waking identity* as parameters (in addition to the constraint itself and to the variable triggering the events of course). When posted, the constraint will be registered to all the events appearing in the events list, along with the waking identity. This integer will be passed to the `update` function whenever one of the events in the list occurs. It allows to dicriminate the event and/or the variable responsible for the wakening, so as to fire a specific rule without having to inspect all the variables to find out the culprit.

A typical use of waking identities is in global constraints that takes an array of variables as parameter. The index of the variable can be associated to the event(s) on which the constraint is suspended and the `update` function may avoid traversing the entire array to compute the propagation.

The use of a waking identity is optional and 0 is assumed (default value) if the parameter is omitted. However, if this feature is used, the identities must be consecutive integers ranging from 0 to $n-1$, and n, the number of distinct wakings, must be passed as an optional parameter (labelled `nb_wakings`) to the `Cstr.create` function. Actually, an array of size n is internally build to record the result of the calls to `update` with each identity. The constraint is solved when all such calls have returned `true` (see 3.3).

3.2.3 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 each update function is called to perform propagation. Propagation may fail by raising an exception or succeed. The propagation of one constraint is also protected against being woken again by itself.

When a constraint is triggered, the update function does not know by which event, nor gets information about the variable responsible of it.

A constraint is woken only once by two distinct events. Note also that the waking queue contains constraints and not variables.

FaCiLe 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 (see 3.3). It can be changed neither when the constraint is posted nor later. Priorities are defined in module `Cstr`: `immediate`, `normal` or `later`.

3.2.4 Constraint Store

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

3.3 User's Constraints

The `Cstr.create` function allows the user to build new constraints from scratch. This function may take up to eight arguments to precisely control the behaviour of the resulting constraint :

```
#Cstr.create;;
- : ?name:string ->
    ?nb_wakings:int ->
    ?fprint:(out_channel -> unit) ->
```

```
    ?priority:Facile.Cstr.priority ->
    ?init:(unit -> unit) ->
    ?check:(unit -> bool) ->
    ?not:(unit -> Facile.Cstr.t) ->
    (int -> bool) -> (Facile.Cstr.t -> unit) -> Facile.Cstr.t
= <fun>
```

However, to define a new simple[3] constraint, very few arguments must be passed to the `create` function as numbers of them are optional (thus labelled) and have default values. Merely the two following arguments are actually needed to build a new constraint by evaluating `Cstr.create update delay`:

- `update` should perform propagation (domains filtering and consistency checks). It must return true iff the constraint is consistent, raise `Stak.Fail` whenever an inconsistency is detected and return `false` otherwise. Its integer parameter should be ignored (as in the first example below) if waking ids are not used (as 0 will consistently be fed as argument).

- `delay` schedules the awakening of the constraint, i.e. the execution of its [update] function. The `delay` argument takes only one argument `ct`, which is the constraint itself. To specify on which events the constraint is to be woken, this function must call `Var.XxxFd.delay` (once or several times) as shown in the example below. This latter function takes an events list, a variable and the constraint `ct` as parameters and returns `()` (unit).

However we recommend to name new constraints and precise their printing facilities, which may obviously help debugging, by specifying the following two 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 values...).

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

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

- `?not` should return the negation of the constraint (which is a constraint itself). It is called when the negation of a reified constraint is entailed, and to access the waking conditions of the negation of a constraint when its reification is posted (and the optional argument `?delay_on_negation` of `Reify.boolean` is set to `true` - which is its default value). Logical operators of module `Reify` also call the `?not` function for the same purpose (see 2.4.4).

To be able to use *waking identities*, their number must be specified:

- `?nb_wakings` default value is 1. This optional argument is used in conjonction with waking identities specified in the `delay` argument. If (contiguous) waking ids 0 to $n-1$ are used, `~nb_wakings:n` must be passed to `Cstr.create`.

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).

[3] That is unreifiable and without the use of waking identities.

- `?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 default and intended behaviours of `init` are a bit intricate when using waking identities. Its detailed use is explained in the next paragraphs with the help of two examples. The default behaviours of `init` is:
  - to call `update 0` and ignores its result, when `nb_wakings` is equal to 1 (which is its default value);
  - to do nothing (`fun () -> ()`) when `nb_wakings` is greater than 1.

  If this is not the desired behaviour, the `init` argument must be specified.

Example of reifiable constraints The example below defines a new constraint stating that variable `x` should be different from variable `y`. This constraint specifies an optional name and an optional printing function. It suspends itself to instantiation events of its two variables (without using any waking identity). Its update function ignores its integer argument (`update _ = ...`) and withdraws the instantiation value of one of its variable in the domain of the other. This constraint is reifiable as the `check` and `not` functions are specified.

Note that no optional `init` function is provided, neither any `nb_wakings` argument: in this case, the default behaviour of `init` will be to call `update 0`. The `init` function is the first function to be called as soon as the constraint is posted, and its usual intended role is to perform an initial propagation and possibly initialize internal data structures of the constraint. This is what happens in this first example. However, if the constraint is suspended on an instantiation event (`XxxFd.on_subst`), and the `update` function relies on the fact that it will only be called when the variable is instantiatied (e.g. without testing that the variable is effectively bound), then the default `init` behaviour is not appropriate. Use a specific `init` function instead by providing this optional argument to `Cstr.create`, as shown in the second example (that uses waking ids).

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 =
    (* The constraint is suspended on the instantiation of x or y. *)
    Fd.delay [Fd.on_subst] x ct;
    Fd.delay [Fd.on_subst] y ct
  and update _ =
    (* If one of the two variables is instantiated, its value is
       removed in the domain of the other variable. *)
    if Fd.is_bound x then
      begin Fd.remove y (Fd.elt_value x); true end
    else if Fd.is_bound y then
      begin Fd.remove x (Fd.elt_value y); true end
    else false
  and check () = (* Consistency check for reified constraints. *)
    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
    | (Unk attr_x, Unk attr_y) when
```

```
    (* If the intersection of domains is empty, the constraint is satisfied. *)
        let dom_x = Var.Attr.dom attr_x and dom_y = Var.Attr.dom attr_y in
        Domain.is_empty (Domain.intersection dom_x dom_y) -> true
    | _ -> raise Cstr.DontKnow
  and not () = fd2e x =~ fd2e y in (* Negation for reification. *)
  (* Creation of the constraint. *)
  Cstr.create ~name ~fprint ~check ~not update delay
```

Let's compile the file:

```
ocamlc -c -I +facile 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.atomic (fun () -> Cstr.fprint stdout diseq)
#  &&~ Goals.fail in
#while (Goals.solve  goal) do () done;;
2: different: 1 <> 2
2: different: 1 <> 3
2: different: 2 <> 3
- : unit = ()
```

Another example to test the reification function `check`:

```
#let x = Fd.create (Domain.create [1;3;5])
#and y = Fd.create (Domain.create [2;4;6]);;
val x : Facile.Easy.Fd.t = <abstr>
val y : Facile.Easy.Fd.t = <abstr>

#let reified_diseq = Reify.boolean (Diff.cstr x y);;
val reified_diseq : Facile.Var.Fd.t = <abstr>

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

Variables `x` and `y` have disjoint domains, so the boolean variable `reified_diseq` is instantiated to 1 as expected.

Example of constraints using waking identities The above example could benefit from the use of waking ids, avoiding the cost of testing which variable has been instantiated within the `update` function. The next example features such a disequality constraint. The `delay` function must now specify a waking id (argument `waking_id`) along with its associated events list and variable. These ids must form an interval ranging from 0 to a given $n-1$, and its size n must be provided to the `Cstr.create` function through its optional `nb_wakings` argument[4]. The

[4] As correctly guessed by the reader, these ids are used to access an internal array.

`update` function now makes use of this information (argument `id`) and performs the appropriate propagation depending on which waking event has occurred. This function must return `true` if the constraint is satisfied for this particular event and `false` otherwise. The constraint will be satisfied only when all the calls to `update 0`, ..., `update (n-1)` have returned `true`.

In this example, we must provide an `init` function as well, because the `nb_wakings` argument is greater than 1 and the default behaviour of `init` is then to do nothing. But the constraint should propagate at post time, so an appropriate `init` function (which incidentally calls the `update` one) is provided.

diffid.ml

```
open Facile
open Easy

let cstr x y =
  let delay ct = (* Ids are associated with waking events. *)
    Fd.delay [Fd.on_subst] x ~waking_id:0 ct;
    Fd.delay [Fd.on_subst] y ~waking_id:1 ct
  and update id =
    begin (* Update function using waking ids. *)
      match id with
        0 -> Fd.remove y (Fd.elt_value x)
      | 1 -> Fd.remove x (Fd.elt_value y)
      | _ -> failwith "Diff_if.cstr: unexpected waking id"
    end;
    true in
  let init () =
  (* Update should be called if x or y is already bound when posting
     the constraint. This is the job of the init function. *)
    if not (Fd.is_var x) then ignore (update 0)
    else if not (Fd.is_var y) then ignore (update 1) in
  (* Creation of the constraint with 2 waking ids. *)
  Cstr.create ~nb_wakings:2 ~init update delay
```

3.4 User's Goals

3.4.1 Atomic Goal: `Goals.atomic`

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

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

```
#let gprint_fd x = Goals.atomic (fun () -> Printf.printf "%a\n" Fd.fprint x);;
val gprint_fd : Facile.Easy.Fd.t -> Facile.Goals.t = <fun>
```

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

```
#let unify_goal x v = Goals.atomic (fun () -> Fd.unify x v);;
val unify_goal : Facile.Easy.Fd.t -> Facile.Easy.Fd.elt -> Facile.Goals.t =
  <fun>

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

Note that this goal is the built-in goal `Goals.unify`.

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

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

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

The FaCiLe implementation of the classic "findall" of Prolog also illustrates the use of `Goals.atomic` to perform side effects: in this case to store all the solutions found in a list. The function `findall` in this example takes a "functional goal" `g` as argument which itself takes the very variable `x` from which we want to find all the possible values for which `g` succeeds; it could correspond to the Prolog term:

```
findall(X, g(X), Sol)
```

```
#let findall g x =
#  let sol = ref [] in
#  let store = Goals.atomic (fun () -> sol := Fd.elt_value x :: !sol) in
#  let goal = g x &&~ store &&~ Goals.fail in
#  ignore (Goals.solve goal);
#  !sol;;
val findall :
  (Facile.Easy.Fd.t -> Facile.Goals.t) ->
  Facile.Easy.Fd.t -> Facile.Easy.Fd.elt list = <fun>
```

We first declare a reference `sol` on an empty list to store all the solutions. Then the simple goal `store` is defined to push any new solution on the head of `sol` – note that we here use `Fd.elt_value v` (see **??**) for conciseness but it is quite unsafe unless we are sure that `v` is bound. The main goal is the conjunction of `g`, `store` and a failure. This goal obviously always fails, so we "ignore" the boolean returned by `Goals.solve`, and the solutions list is eventually returned.

The main point when creating goals is to precisely distinguish the time of *creation* of the goal from the time of its *execution*. For example, the following goal does not produce what might be expected:

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

The `min` and `max` of the variable `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 within the goal. Then the latter must return the disjunction, which cannot be done with a simple call to `Goals.atomic`; function `Goals.create` (described in the next section) must be used instead.

3.4.2 Arbitrary Goal: `Goals.create`

The function `Goals.atomic` does not allow to construct goals which themselves construct new goals (similar to Prolog clauses). The `Goals.create` function "goalifies" a function which must return another goal, possibly `Goals.success` to terminate.

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

```
#let min_or_max v =
#  Goals.create
```

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

The other difference between `Goals.create` and `Goals.atomic` 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.create`. In the previous example, we use `()`.

`Goals.create` allows the user to define recursive goals by a mapping on a recursive function. In the next example, we iterate a goal non-deterministically on a list. Note that this goal is equivalent to the built-in goal `Goals.List.exists`.

```
#let rec iter_disj fgoal list =
#  Goals.create
#    (function
#         [] -> Goals.success
#       | x::xs -> fgoal x ||~ iter_disj fgoal xs)
#    list;;
val iter_disj : ('a -> Facile.Goals.t) -> 'a list -> Facile.Goals.t = <fun>

#let gprint_int x = Goals.atomic (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.create_rec`

FaCiLe provides also a constructor for intrinsically recursive goals. Expression `Goals.create_rec f` is similar to `Goals.create 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.create_rec (fun self -> 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-deterministically instantiates a variable is written using `Goals.create_rec`:

```
#let indomain var =
#  Goals.create_rec ~name:"indomain"
#    (fun self ->
#      match Fd.value var with
#        Val _ -> Goals.success
#      | Unk attr ->
#         let dom = Var.Attr.dom attr in
#         let remove_min =
#           Goals.atomic (fun () -> Fd.refine var (Domain.remove_min dom))
#         and min = Domain.min dom in
#         Goals.unify 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 does 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.

3.5 Backtrackable Invariant References – BIRs

FaCiLe provides through the module `Invariant` some features to handle data-structures which are functionnally dependant between each other. These *invariants* are directly derived from the work of [8], although they are meant to be used within CP search goals. So they'll be called *backtrackable invariant references* or BIRS in the sequel, as their values are restored upon backtracks.

An invariant is either:

- a constant,
- or a mutable value,
- or the result of a function applied to other invariants,
- or an attribute of any dynamic data-structure, e.g. the maximal value of a finite domain variable.

FaCiLe can provide efficient handling of the dependencies between BIRs in order to keep them updated. For example, if an integer (or floating point) BIR i is defined as the sum of n others $i_1, \ldots, i_n$, the change of the value of one of $i_1, \ldots i_n$ will be taken into account in constant time to update i.

The main original use of invariants proposed in [8] is within local search algorithms. In our context, it can be used also to compute a heuristic criterion used during search. The implementation of BIRs in FaCiLe is fully compatible with backtrack.

In the following, we call BIR a mutable value and *invariant* the relation (a functional equation) between BIRs.

3.5.1 Type, creation, access and modification

BIRs of FaCiLe are polymorphic so you can handle any data-structures with them. A BIR may be mutable or not and this property is handled by the typing:

- a mutable integer BIR has type `(int, setable) Inv.t`,
- whereas a non mutable float BIR has type `(float, notsetable) Inv.t`.

However, shortcuts with only one type parameter are defined in module `Invariant` to simplify the writting of BIRS: `'a setable_t` and `'a unsetable_t`.

We show in the following example how to create, access and modify a BIR:

```
#let x = Invariant.create 1729 and y = Invariant.constant 3.14;;
val x : int Facile.Invariant.setable_t = <abstr>
val y : float Facile.Invariant.unsetable_t = <abstr>

#(Invariant.get x, Invariant.get y);;
- : int * float = (1729, 3.14)

#Invariant.set x 1730;;
- : unit = ()

#Invariant.get x;;
- : int = 1730
```

Like finite domain variables, BIRs can be named thanks to an optional string argument (`?name`) and feature a unique integer identity (accessible with function `Invariant.id`).

3.5.2 Operations

FaCiLe provides basic arithmetic operators on integer BIRs. These functions are completed by primitives working on array of BIRs (submodule `Invariant.Array`).

The following table gives the time and space complexity of the basic invariants :

Invariant	Time	Memory
$s = \sum_{i=1}^{n} x_i$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$p = \prod_{i=1}^{n} x_i$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$m = \min_{i \in [1,n]} x_i$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
$i = \text{argmin}_{i \in [1,n]} x_i$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

```
#let a = Array.map Invariant.create [|1;2;3;4|];;
val a : int Facile.Invariant.setable_t array =
  [|<abstr>; <abstr>; <abstr>; <abstr>|]

#let s = Invariant.sum a;;
val s : int Facile.Invariant.unsetable_t = <abstr>

#Invariant.get s;;
- : int = 10

#Invariant.set a.(3) 8;;
- : unit = ()

#Invariant.get s;;
- : int = 14
```

The library also provides generic wrappers (`unary`, `binary` and `ternary`, for functions with arity up to tree) for BIRs which allow the user to transform any function working on the type α into a function working on an α BIR:

```
#let x = Invariant.create ~name:"x" 2.71;;
val x : float Facile.Invariant.setable_t = <abstr>

#let y = Invariant.unary ~name:"log" log x;;
val y : float Facile.Invariant.unsetable_t = <abstr>

#Invariant.fprint stdout y; Invariant.get y;;
log(x)- : float = 0.996948634891609564

#Invariant.set x 8.0; Invariant.get y;;
- : float = 2.07944154167983575
```

These wrapped functions can be named with an optional string argument.

3.5.3 Domain access

In order to implement computation of heuristic criterion, it is required to be able to translate attributes of finite domain variables (of type `Fd.t` or `SetFd.t`) into invariant references. These functionalities are listed in submodules `Invariant.Fd` and `Invariant.SetFd`.

For example, the heuristic criterion which selects the variable with the smallest domain is easily computed as follows :

```
#let best = fun vars ->
#  Invariant.Array.argmin (Array.map Invariant.Fd.size vars);;
val best :
  Facile.Invariant.Fd.fd array ->
  (int -> 'a) -> int Facile.Invariant.unsetable_t = <fun>
```

Part II

Reference Manual

3.6 Module Easy

`Easy` is a module that the user is strongly advised to open in order to facilitate access to FaCiLe (unless names clash with other open modules). It simply defines aliases to values and types of other modules:

- All the infix operators from `Arith`, `Goals` and `Reify`
- Frequently used mapping functions of `Arith`: `i2e` and `fd2e`
- Type of finite domain variables from `Var`: `concrete_fd = Unk of Fd.attr | Val of Fd.elt`
- 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] Nicolas Barnier. *Application de la programmation par contraintes à des problèmes de gestion du trafic aérien*. PhD thesis, Institut National Polytechnique de Toulouse, December 2002. `www.recherche.enac.fr/opti/papers/thesis/`. iii

[2] Nicolas Barnier and Pascal Brisset. FaCiLe: a Functional Constraint Library. *ALP Newsletter*, 14(2), May 2001. iii

[3] 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. 18

[4] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Fourteenth International Joint Conference on Artificial Intelligence IJCAI'95*, volume 1, pages 607–615, Montral, Qubec, Canada, August 1995. Morgan Kaufmann. 22

[5] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, 1973. 17

[6] Serge Le Huitouze. A new data structure for implementing extensions to Prolog. In P. Deransart and J. Małuszyński, editors, *PLILP'90, LNCS 456*, pages 136–150. Springer-Verlag, 1990. 10

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

[8] Laurent Michel and Pascal Van Hentenryck. Localizer: A modelling language for local search. In *Proceedings of the Third Conference on Principles and Practice of Constraint Programming*, 1997. 37

[9] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996. 18