FaCiLe: a Functional Constraint Library

Pascal Brisset[1] and Nicolas Barnier[2]

[1] École Nationale de l'Aviation Civile, Toulouse, France
[2] Centre d'Études de la Navigation Aérienne, Toulouse, France
{brisset,barnier}@recherche.enac.fr

Abstract. FaCiLe is an open source constraint programming library over integer finite domain written in OCaml, a functional language of the ML family. It offers all usual constraint system facilities to create and handle finite domain variables, arithmetic constraints (possibly non-linear), built-in global constraints and search goals. FaCiLe allows as well to build easily user-defined constraints and goals from scratch or by combining simple primitives, making pervasive use of 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 polymorphic type inference and strong typing discipline, high level of abstraction, generic modules and object system, as well as native code compilation efficiency, garbage collection and replay debugger. All these features allow to prototype and experiment quickly: modelling, data processing and interface are implemented in the same powerful language with a high level of safety.

1 Introduction

When designing a Constraint Programming (CP) system, several committing choices have to be made. Here we try to give convincing arguments to justify the ones we have made in the development of FaCiLe with the functional language Objective Caml [12].

First, we have deliberately avoided the "Yet Another Language" approach: we do not want to reinvent the wheel and restart the language designers work and compromises. From this point of view, we differ from Oz [20] or Claire/Choco [10] systems. Among CP dedicated languages, a modelling language like OPL [24] plays a different role: it may be viewed as an upper layer over some CP system which still needs to be accurately chosen. We found that the most suitable approach would be to design a *library* for a programming language which has to meet the following requirements: efficiency, portability and availability ("open source").

Secondly, a choice must be made between "low-level" imperative languages and "high level" applicative languages. In the first class, we find the unavoidable and widespread C++ for which the world leading commercial system Ilog Solver [21] has been developed. The main advantage of such a language is execution speed efficiency but the list of drawbacks is too long for our taste: tedious

and error-prone manual memory management, endless problems with pointers, poor typing, lack of data-structures, weak modularity, verbosity... In the second class, we find logic programming and functional programming.

Logic programming historically was the best candidate for the Constraint (Logic) Programming paradigm [23]: constraints are predicates and non-deterministic search is inherently provided in Prolog. Unfortunately, if this adequation works well for toy examples, it has a significant cost in real-size programming: data processing, writing of efficient propagation algorithms for global constraints, displaying results, etc. are not easy tasks within logic programming. The lack of typing and compile-time checks in common Prolog systems[1] is also a reason to avoid them in large projects development for safety and robustness arguments.

Functional programming does not offer the adequate logic for constraints but avoids the previously mentioned drawbacks of "low level" languages and provides several suitable features: strong semantics, higher-order, automatic memory management, modularity with genericity (functors), strong typing with polymorphism. Last but not least, the OCaml implementation offers well-documented libraries, portability and efficiency [1, 13, 15]. All these suitable features have led us to use OCaml both as a source language for writting constraint programs *and* for the implementation itself of the library. A few constraint frameworks have already been designed over functional languages such as Lisp like the seminal object oriented PECOS [16] system or SCREAMER [18]. Our approach is closer to the work presented in [7] which uses higher-order support and exception handling of ML to provide programmable search engines and backtrack.

This paper is divided in two main parts: we first introduce in section 2 the main features of FaCiLe, starting with a small standard example and focusing on the usage of the library, notably on the design and expressiveness of FaCiLe search goals; then we explore the architecture and implementation issues in section 3. Section 4 features a small benchmark of FaCiLe based on classic examples from the litterature. Conclusion and future work are eventually presented in section 5.

2 A Tour of FaCiLe

FaCiLe is divided into numerous modules which structure the library and its usage: `Domain` for domain handling, `Fd` for finite domain variables, `Arith` for arithmetic expressions, `Cstr` for constraint handling, `Goals` for search specification and control, etc. The main type in each of these modules is named '`t`' and we will refer to it with its qualified "dot" notation: `Module.t`.

We first give a taste of FaCiLe through the pervasive "magic sequence" example, quickly reviewing a few features of the library. The next section introduces the arithmetic and global constraints of FaCiLe. The third part is devoted to a thorough exploration of the clean semantics and expressive power of FaCiLe search and optimization procedures.

[1] Except recent systems like Mercury [22] and Gödel [8].

2.1 A Short Example

Programming with FaCiLe follows the classic CP scheme as illustrated in the example below that (naively) solves the *magic sequence* problem (a magic sequence is a sequence of n integers $X = (x_0, x_1, ..., x_{n-1})$ such that 0 will appear in the sequence x_0 times, 1 will appear x_1 times etc.): 1. definition of domains and variables – 2. definition and posting of the constraints – 3. search, possibly with optimization.

```
let xs = Fd.array n 0 (n-1);;
let is_equal_to i x = fd2e x =~~ i2e i;;
Array.iteri
  (fun i xi ->
    let cardi = Arith.sum (Array.map (is_equal_to i) xs) in
    Cstr.post (fd2e xi =~ cardi)) xs;;
Goals.solve (Goals.Array.forall Goals.indomain xs);;
```

We first define an array `xs` of n variables ranging from 0 to $n-1$, using the function `Fd.array`. Then we iterate on `xs` to post the family of constraints specifying the number of occurences of a given integer in the array: $|\{x \in xs \ \ s.t. \ \ x = i\}| = x_i$. The iterator is a simple call to the standard library function `Array.iteri` which sequentially applies a function on each index and element of an array. Here, the cardinality constraints are computed thanks to the reified equality operator `=~~` which returns a boolean expression. To perform this step, we define the function `is_equal_to` which is then *mapped* on all the element in `xs` at each iteration. Then the sum of all the element of this new array is obtained with the operator `Arith.sum`. The cardinality constraint (of type `Cstr.t`) is eventually posted to the constraint store with the function `Cstr.post`.

Note that operators are suffixed with character '~' (`"~~"` for reified ones) and explicit conversion functions (`i2e`, i.e. "integer to expression" and `fd2e`, i.e. "FD variable to expression") must be used to build an arithmetic expression because of the strong typing discipline of OCaml which does not allow overloading or implicit casting. In compensation, the behaviour of expressions is simply and soundly specified: the result is exactly what expected and errors are caught at compile time.

Finally, a goal is defined by an iteration of a standard labelling built-in function on each variable (`Goals.indomain`). Another polymorphic iterator is here used, `Goals.Array.forall` with type:

$$(\alpha \rightarrow Goals.t) \rightarrow \alpha \ array \rightarrow Goals.t$$

α being a type parameter (standing for any type) and `Goals.t` the type of goals. Only a function working on the elements of the array and returning a goal needs to be supplied to the iterator to build the conjunction of the corresponding goals in increasing order of the indices. Then, a solution is searched by passing the goal to the `Goals.solve` function. `Goals.solve` has type $Goals.t \rightarrow bool$ and returns `true` if the goal succeeds and `false` otherwise.

This basic example might not feature fewer lines of code than with an *ad hoc* language, but its concision is noticeable for a library-based scheme. The ability of OCaml to handle functions as first class objects and the iterator-oriented programming style, together with the type inference, allow to define and compose functions in a concise way.

2.2 Constraints

FaCiLe features classic linear and non-linear arithmetic constraints, global constraints such as the "global cardinality" constraint and the novel "sort" constraint. Reified constraints and logical operators over constraints are provided as well. Moreover, FaCiLe allows the user to easily define its own constraints through a higher-order interface and fine control of waking conditions.

Arithmetic Constraints FaCiLe provides standard linear constraints and non-linear constraints such as exponentiation, modulo or absolute values. For example, an inequation such as:

```
let x = Fd.interval (-2) 6 and y = Fd.interval 4 12;;
let xe = fd2e x and ye = fd2e y;;
let expr = i2e 10 *~ (xe **~ 2) *~ ye +~ i2e 4 *~ ye;;
let ineq = expr >=~ i2e 4300;;
```

once posted to the constraint solver (with `Cstr.post`), yields a single solution: $x = 6$ and $y = 12$.

As mentioned earlier, finite domain variables of module `Fd` (type `Fd.t`) are not arithmetic expressions (type `Arith.t`) and conversion functions must be used to transform a variable or an integer into an expression (`fd2e` and `i2e`), or to transform back an expression into a new variable (`e2fd`). Arithmetic constraints (equality, difference, strict and non-strict inequality) are infix operators that takes two arithmetic expressions and returns a constraint (type `Cstr.t`). Moreover, all the operators of module `Arith` must be suffixed with character `'~'`. This is due to OCaml strong typing discipline that leads to clearly make the difference between these various objects and does not allow to mix them without being aware of the desired behaviour. However, the verbosity cost often pays off by avoiding spurious confusions between variables and expressions – for example trying to instantiate an expression with an integer value. The combination of overloading with implicit casting indeed often tends towards hard to debug code.

Global Constraints FaCiLe provides some well-known global constraints like *all different* or *element*, and the novel *sort* constraint [6] which optimally (complexity and consistency) narrows the variables bounds of two arrays, stating that the second one must be equal to the sorted first one. A *global cardinality constraint* is also available (`Gcc.cstr`) and the magic sequence example can be improved by making use of it, as it triggers more domain reductions than its reified arithmetic counterpart:

```
let card_values = Array.mapi (fun i x -> (x, i)) xs;;
Cstr.post (Gcc.cstr xs card_values);;
```

This constraint takes as arguments an array of variables `xs` and an array of couples `(card, value)`, where `card` is a variable and `value` an integer, to enforce that `card` variables in `xs` are equal to `value`. Note that the cardinality and difference constraints take an optional argument to finely specify the consistency level / time cost trade-off; the sort constraint optionally takes the permutation array as argument.

A simple interface with higher-order functionals allows the user to define its own (possibly reifiable) constraints. Implementation of complex filtering algorithms is eased by the expressiveness and safetiness of OCaml: e.g. the intricate sort constraint (numerous complex data structures and incremental maintenance) has been efficiently implemented in a short time and very straightforwardly (230 lines of code).

2.3 Search

Goals To control the search for solutions, FaCiLe provides built-in goals and functions to create simple or recursive user-defined goals which can be combined by conjunction (`&&~` operator) or disjunction (`||~`). All the functions, constants and operators related to goals are gathered in module `Goals` and the type of goals is `Goals.t`.

For example, the enumeration of all solutions such that a given goal `g` succeeds would trivially be written in FaCiLe as follows:

```
let all_solutions g = g &&~ Goals.fail
```

The side effects needed to print or store the solutions found, as in the classic Prolog goal `findall`, can be performed by using the FaCiLe primitive `Goals.atomic`. The following function stores each solution in a list; it takes a "functional goal" `g` as argument which itself takes the variable `x` from which we want to find all the possible values such that `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.int_value x :: !sol) in
  let goal = g x &&~ store &&~ Goals.fail in
  ignore (Goals.solve goal);
  !sol
```

First, a reference (`sol`) on an empty list is declared to store all the solutions. Then the `store` goal is defined to push any new solution on the head of `sol`; it uses `Goals.atomic` which "goalifies" any function that does not return a goal (or anything else), but only performs side effects. 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` that informs whether the goal has succeeded or not, and the list of solutions is eventually returned.

A more general version of `findall` that returns any kind of solutions (possibly not integers) can be very easily written as well. While the previous `findall` function had type:

$$(Fd.t \rightarrow Goals.t) \rightarrow Fd.t \rightarrow int\ list \tag{1}$$

this new polymorphic `findall` function would take a goal and a function extracting the solution from the involved variables. Therefore its type would be:

$$Goals.t \rightarrow (unit \rightarrow \alpha) \rightarrow \alpha\ list \tag{2}$$

α being a type parameter. In contrast to type 1, type 2 shows that no variable is given anymore to `findall`, but the goal and storing function can refer to them with a closure. Note that the `unit` predefined type is the type that has only one value, namely `()`; it is here used as a "dummy" argument to defer the evaluation of a function that should not be executed at the time of its creation but at a later given time (OCaml has a strict evaluation scheme).

This general `findall` function can be written as follows:

```
let findall g extract =
  let sol = ref [] in
  let store = Goals.atomic (fun () -> sol := extract () :: !sol) in
  let goal = g &&~ store &&~ Goals.fail in
  ignore (Goals.solve goal);
  !sol
```

Such flexible reusable functions are very easily defined with FaCiLe, due to the polymorphic typing and higher-order functionals of OCaml.

Thanks to higher-order and the clean semantics of goals in FaCiLe, more flexibility can be achieved like, for example, to use the result of the `findall` function as the argument of a continuation goal, which is passed as the third parameter to `findall`. This latest `findall` function does not return a list anymore as shown with type 1 or 2, but it builds a new goal that may be combined with other ones to yield a complex search procedure:

```
let findall g extract cont =
  let sol = ref [] in
  let store = Goals.atomic (fun () -> sol := extract () :: !sol) in
  (g &&~ store &&~ Goals.fail) ||~ (Goals.create cont !sol)
```

The `cont` higher-order argument is simply a function working on a list (whose elements are of the same type α than those returned by `extract`) and returning a goal. To build this continuation goal, we here use the built-in function `Goals.create`, thoroughly explained later, that allows to create a goal that itself returns another goal. Hence, this new `findall` function has type:

$$Goals.t \rightarrow (unit \rightarrow \alpha) \rightarrow (\alpha\ list \rightarrow Goals.t) \rightarrow Goals.t \tag{3}$$

Implementing sophisticated search goals that need to exchange data is quite natural in Prolog: logic variables are simply shared between goals and unification does the rest. As illustrated in the following scheme, "out" parameters of the first goal are just taken as "in" parameters of the second one (Prolog syntax):

$$g_1(In, Out), g_2(Out)...$$

With FaCiLe, only finite domain variables are suitable objects to follow a similar scheme, but integers obviously lack expressive power. However, we can use the higher-order continuation scheme to overcome this issue, such that the previous conjunction can be translated into the following composition (functional syntax):

$$g_1\ in\ (\lambda out\ (g_2\ out))$$

Goal g_1 actually only takes the continuation as an extra argument – as seen in the last `findall` example – and any result of the computation of g_1 can be passed to g_2.

Another workaround to address this issue would be to share global variables between goals. Unfortunately, this scheme does not get along well with the non-deterministic control of goals and backtracking (though FaCiLe provides backtrackable references). So we believe that continuation style encoding of complex search goals is a suitable and handy paradigm that integrates well in functional programming and achieves an expressiveness close to Prolog systems one.

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 that 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 (such as `Array.iteri` used in the first example), FaCiLe provides iterators for arrays and lists: `Goals.Array` and `Goals.List` modules of FaCiLe allow to construct conjunctions and disjunctions of goals.

The polymorphic function `Goals.Array.forall` uniformly applies a goal to every element of an array:

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

and has the following type:

$$(\alpha \rightarrow Goals.t) \rightarrow \alpha\ array \rightarrow Goals.t$$

The labelling of an array of variables is the iteration of the instantiation of one variable with `Goals.indomain`:

```
let labeling_array = Goals.Array.forall Goals.indomain;;
```

and this function obviously has type $Fd.t\ array \rightarrow Goals.t$. A matrix is an array of arrays; following the isomorphism, labelling of a matrix simply is a composition with the array iterator, which can be elegantly written in OCaml:

```
let labeling_matrix = Goals.Array.forall labeling_array;;
```

with type $Fd.t\ array\ array \rightarrow Goals.t$.

The ability to partially apply function and handle them as first class objects yields very concise and simple code to build search goals that iterate over complex data structures. Together with optional arguments, soundly integrated within OCaml type system, these features offer a very user-friendly and consistent interface, more efficient and safer than "macros" or Prolog goals wich have poor support for variables handling, and less verbose and tedious than writting new objects (with an OO language) for example. We believe that programming languages like OCaml have enough expressive power (with higher safety) to meet the aim of modelling languages such as OPL, while having the efficiency and processing/interface abilities of main-stream imperative languages.

Recursive goals To define a recursive goal, i.e. a goal that returns another goal, FaCiLe provides the `Goals.create` primitive which has the following type:

$$(\alpha \rightarrow Goals.t) \rightarrow \alpha \rightarrow Goals.t$$

`Goals.create f a` builds a goal which will apply `f` to `a` when it will be evaluated, i.e. goals creation implements some kind of deferred (or lazy) evaluation (as previously mentioned, OCaml has a strict evaluation scheme). Note that this value (`a`) has a polymorphic type (α) and so can be used to pass several arguments (with a tuple) – or the function itself may contain a closure.

The following example illustrates the use of such recursive goals. This piece of code disjunctively explores a list with a continuation applied to one element and to the rest of the list, preserving the original order of the elements of the list:

```
let rec delete l cont =
  Goals.create
    (function
       [] -> Goals.fail
     | x :: xs ->
         cont x xs ||~ delete xs (fun y r -> cont y (x :: r)))
    l;;
```

We here use an extensional definition by pattern-matching for the function passed to `Goals.create` (starting with the OCaml keyword `function`): if the list is empty, we simply fail; otherwise we build the disjunction that either applies `cont` to the head `x` and the rest of the list `xs` (`cont` must return a goal) or recursively tries the next element, keeping the integrity of the list by putting back `x` on the top of the rest of the list during the application of `cont` in the next iteration. Note that `::` is the OCaml constructor for lists, which may be use either for building a new list or for pattern-matching.

This last example involving a recursive goal shows well how versatile can be the FaCiLe goals module. Due to the matchless support for higher-order provided

by OCaml, FaCiLe search procedures achieve a degree of expressiveness close to Prolog goals. This feature combined with the processing ability and safety of the host language lead to very fast and concise prototyping of complex search strategies.

Optimization FaCiLe provides also a `Goals.minimize` primitive which, once given a goal and a cost (a finite domain variable), builds a new goal that, when executed, runs a branch and bound algorithm. `Goals.minimize` has the following type:

$$Goals.t \rightarrow Fd.t \rightarrow (int \rightarrow unit) \rightarrow Goals.t$$

The first argument obviously is the goal to solve and the second one is the cost variable. The third one is a function applied to the instantiation value of the cost whenever a solution is found. It may be used to print the solution and its cost or store them with a reference.

As it simply builds a new goal, `Goals.minimize` has a clean semantics, such that FaCiLe optimization procedures can be safely nested and finely controlled to yield complex strategies.

3 Implementation

The implementation is as naive as possible in order to keep the library short and maintainable (4000 lines of code). FaCiLe is entirely written in OCaml, even low-level critical parts like domain manipulations or propagation over arithmetic expressions. The advantages are the conciseness and robustness of the result. The drawbacks may be low efficiency compared to hard-coded libraries.

We give in this section a description as precise as possible within the available place. The description follows the architecture of the library which is reflected in its decomposition into modules. This decomposition and the dependences between the modules are displayed in figure 1.

3.1 Stack

Non-deterministic search is classically handled by a stack in FaCiLe. The `Stack` module provides one (and only one) global abstract stack. This stack is structured into *levels*. A level contains a success continuation (i.e. a list of *goals*) and a trail. A trail (the failure continuation) is a list of *things to undo*. This is implemented as a list of closures, providing the most general mechanism. It may be considered too expensive for the majority of trailings (which are reduced to the restoration of a pointer) but our time profilings does not confirm this intuition. The bad point surely is the heavy memory consumption of this technique and we are waiting for the annouced OCaml memory profiler to trace it.

A level is created to handle one choice-point. It is created when it is pushed on the stack. It is used while backtracking: closures of the *undo* list are called and the stored success continuation is returned (usually to replace the current one).

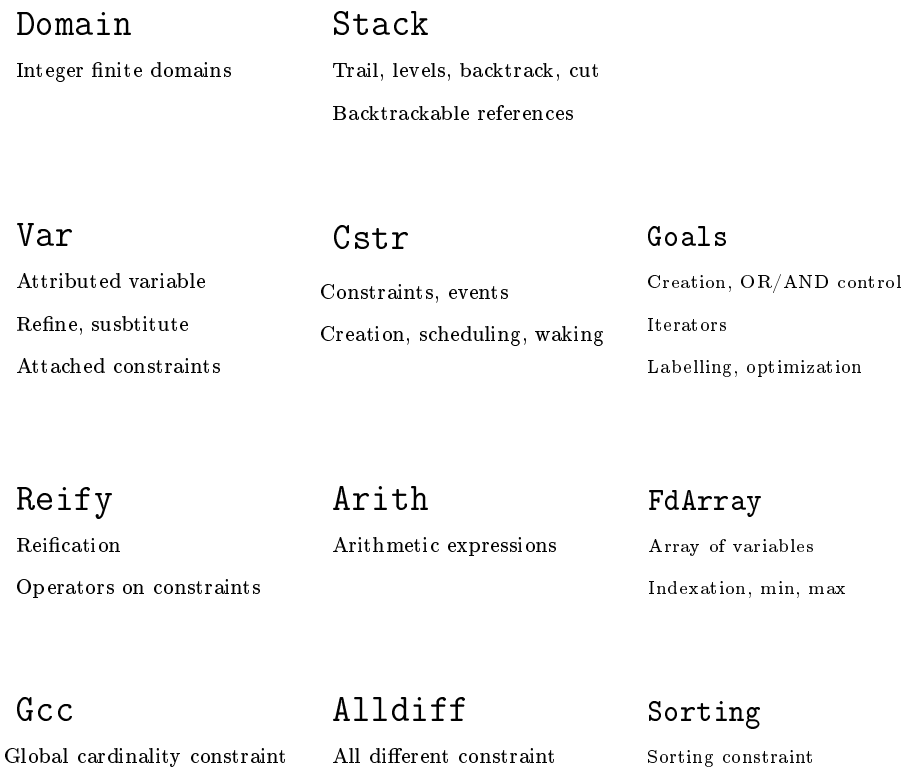

Fig. 1. Architecture of the FaCiLe library

A level can be *cut*; it is then marked (tagged) inactive and further backtracks ignore it.

The module provides abstract backtrackable polymorphic references built on top of the basic mechanisms (the undo closure related to an assignment is an assignment with the old value).

Eventually, this module provides the failure mechanism simply implemented with the exception handling of OCaml.

3.2 Constraints

The module `Cstr` offers several abstract datatypes, *priority*, *event* and *constraint*, and provides functions to handle constraints: creation, scheduling and waking.

A constraint is essentially a callback function (named *update* in FaCiLe terminology) which is called when the constraint is woken. This callback is responsible for checking the satisfiability and consistence of the relation between the involved variables, refining the domains by removing the non consistent values

and possibly raising a failure. A consistent constraint is tagged *solved* and is no longer considered. All these processings are the responsability of the designer of the constraint.

To be scheduled for waking, a constraint has to be *delayed* on an event. An event (abstract type) is a collection of constraints (we say that constraints are *registered* with the event) which may be woken when the event is scheduled. FaCiLe attaches events to variables (see below) and to the OR control. This particular event is scheduled each time a choice point is created.

A scheduled constraint is woken according to its priority: constraints are stored in a queue and the following rule is ensured: a constraint c of a given priority is woken after all the constraints of higher priority scheduled before the waking of c. A tag mechanism ensures that a constraint cannot be scheduled twice in the same time, even by two different events.

This module also manages a global store of all the active (not satisfiable) constraints. This store is handled in such a way that it does not interfere with the garbage collector: *weak* pointers of OCaml ensures that a constraint no longer used by other data-structures may be automatically removed from the store by the GC (i.e. the store is not a root for useful data).

The stack and the constraint store constitute one single global state. However, encapsulation of this state would be easy to implement through a funtorization of the whole library .This solution has been tested and finally abandoned for the sake of simplicity and because we did not encounter convincing programming examples where concurrent states were required.

3.3 Goals

The search control is implemented on top of the stack, coding the success continuation as a list of *goals*, an abstract data type provided by this module. Like constraints, a goal is essentially a callback, a function called when the goal is activated which possibly returns other goals. Goals of the success continuation are called one after the other until a failure (an exception) is raised; in this case the last choice-point is popped and the saved continuation is restored.

Goals are built from atomic ones (closures), binary operators (conjunction and disjunction), bounded loops (*for to*), existential quantification and iterators on data-structures. Functional programming allows the user to compose iterators, simply mapping them on its data-structures.

This search-goal approach is inherited from logic programming and has been kept for simplicity and efficiency reasons. The search could be made more generic if the state restoration mechanism (currently a stack) is modified.

The other feature of the module is an optimization goal which implements a branch and bound search. With *restart* mode, the search restarts at the root of the tree each time a solution is found (cut is then used). With the *continue* mode, only the constraint bounding the cost variable is updated *in place* each time a solution is found.

3.4 Finite Domains

Domains are represented as minimum and maximum values, size and list of intervals (it was the coding used in the early finite domain library of ECLiPSe). Modification on domains are functional (no in-place modification) so (transparent) structure sharing is essential. Profiled execution shows that this coding is correct for time processing. For this reason, we did not try to implement more sophisticated data-structures (binary trees would give logarithmic access to elements while the current solution provides linear complexity for most of the operations).

Note that this non-destrutive coding may facilitate state restoration required by sophisticated search strategy[3]

3.5 Finite Domain Variables

Variables are standard attributed variables as described in [11]. The attribute contains the domain and the constraints related to the variable. These constraints are listed according to their associated *waking* events. FaCiLe currently offers four kinds of event for variable modification: instantiation to an integer value, change of lower bound, upper bound or any value in the domain. Because no modification takes place inside domains (functional data-structure), they can be shared among several variables.

FaCiLe currently offers only variables over integer domains but the described mechanisms are not specialized to integers and abstraction facilities of the language allows to easily generalize these variables to any domains (interval of floating point numbers, sets, etc.).

3.6 Reification

The `Reify` module defines the relation between a constraint and a boolean variable (actually an 0-1 integer variable). This relation is itself implemented as a constraint. This mapping allows to define logical operators over constraints, translating constraint expressions into arithmetic expressions.

To be reifiable, a constraint requires a supplementary callback responsible for checking the relation without doing any propagation. It also needs a "not" method which returns the negation of the constraint.

Once again, strong typing of the language forbids to mix arithmetic expressions and constraints without explicit conversion (provided by this module). We consider it as an advantage because this operation is not given for free (reification is a constraint) and the user has to be aware of it.

3.7 Arithmetic Expression

Classic linear and non-linear arithmetic constraints are available in FaCiLe, including exponentiation, division, modulo and absolute value. Furthermore, when an arithmetic constraint is posted, FaCiLe compiles and simplifies ("normalizes")

the expression as much as possible so that variables and integers may be scattered inside an expression with no loss of efficiency: the adequate constraints, i.e. the ones that trigger the most domain reductions, will be chosen by the solver.

Moreover, internal intermediate variables used to simplify expressions containing absolute value, exponentiation etc. are collected and hashed to detect redondancy and use as few extra variables as possible.

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 mechanism more efficient than the one used for standard sums.

FaCiLe provides these arithmetic constraints with a high degree of robustness: for example, the implementation of the magic sequence problem included as an example with the standard current distribution works fine with more than 4500 variables on a PC running Linux with 512MO of RAM – much more than all systems benchmarked in [4] if enough memory is available.

3.8 Global Constraints

FaCiLe currently provides three kinds of global constraints. The `Alldiff` constraints is implemented with a bin-matching algorithm [9] which ensures that the constraint is satisfiable in complexity $n^{\frac{5}{2}}$ in worst case. The cardinality constraint (`Gcc`) implements the algorithm described in [17] and proposes three levels of propagation : satisfiability, removing of non consistent values from variables, removing of non consistent values from cardinals. The *sort* constraint implements the algorithm described in [6] (complexity $n\log(n)$).

4 Examples and Benchmark

We give in this section other samples of code and results obtained with FaCiLe for some benchmarks. Times are compared with those of ILOG Solver and ECLiPSe.

Magic Sequence This classic problem is described in the early section 2.1. Results given in table 1 show the robustness of the system and especially of the global cardinality constraint.

N Queens We give here a formulation with FaCiLe of this classic example: we define an array of variables (`Fd.array`) and two auxiliary arrays obtained by shifting the first one by addition (`+~`) and substraction (`-~`).

```
let queens = Fd.array n 0 (n-1) in
let shift op = Array.mapi (fun qi -> Arith.e2fd (op (fd2e qi) (i2e i))) in
let diag1 = shift (+~) queens and diag2 = shift (-~) queens
```

Global `Alldiff` constraints are then set on the three arrays:

```
Cstr.post (Alldiff.cstr queens); Cstr.post (Alldiff.cstr diag1); ...
```

To implement the strategy which first selects the variable with the smallest domain and the smallest minimum, we use the `choose_index` function which returns a selector (optional argument `~select` of `forall`) according to a comparison function which will be applied to unbound variables only.

```
let h a = (Var.Attr.size a, Var.Attr.min a) in
let min_min = Goals.Array.choose_index (fun a1 a2 -> h a1 < h a2) in
let labeling = Goals.Array.forall ~select:min_min Goals.indomain
```

Table 1 gives some results with this program; the number of bactracks, not displayed here, is always very small (a dozen) with the given strategy.

Golomb Ruler A Golomb ruler is a set of integers (marks) $a_1 < ... < a_k$ such that all the differences $a_j - a_i$ for $i < j$ are distinct. We may assume $a_1 = 0$. Then a_k is the length of the Golomb ruler. For a given number of marks, we are interested in finding the shortest Golomb rulers [5].

Results shown in table 1 are obtained with a basic model and a straightforward strategy.

Social Golfer Problem The problem consists in trying to schedule m groups of n golfers over a number of w weeks, such that no golfer plays in the same group as any other golfer twice (i.e. maximum socialisation is achieved). Default values (problem `010` in CSPlib, `www-users.cs.york.ac.uk/~tw/csplib`) are $m = 8$ groups of $n = 4$ golfers. Best known results for this problem are collected by Warwick Harvey and available at `www.icparc.ic.ac.uk/~wh/golf`. The `5-3-7` instance is also known as the "Kirkman's Schoolgirl Problem"[19].

We have tried three models for this problem using respectively the global cardinality constraints, the sorting constraints and sets. Table 1 gives some results for the three models. First column is the standard benchmark of 32 golfers over 9 weeks. Last column is the result for the last case of Warwick Harvey's table; note that this instance is easily solved. Fourth column corresponds to the Kirkman's Schoolgirl instance while second and third columns are subproblems of the latter: they are surprisingly harder although less constrained.

5 Conclusion

We have described in this paper a new functional constraint library written in OCaml. We have given arguments and examples to show that the choice of a high-level functional language is a good compromise among other solutions (modelling language, library for an imperative language, high-level logic programming system) to get an expressive, precise and efficient system. We agree on this point with the last proposal of Van Hentenryck with his Modeler++[14] but with a safer though less trendy host language.

We have chosen a straightforward implementation with as many abstractions as possible (abstract data-types, modularity) in the same vein than the standard

Magic	100			200			400			800			1600		
	0.25	0.05	1.5	1.0	0.15	6.1	4.9	0.48	26	21	2	♯[a]	89	7.8	♯
Queens[b]	16			32			64			128			256		
	0.03	0.03	10	0.07	0.04	∞	0.26	0.06	∞	1.2	0.2	∞	6.0	0.74	∞
Golomb	6			7			8			9			10		
	0.04	0.04	0.18	0.33	0.11	1.2	3.3	0.75	11	30	7.0	99	280	65	820
Golf[c]	8-4-9			5-3-4			5-3-6			5-3-7			8-8-9		
`gcc`	2.0	0.7	5.3	370	100	373	1660	400	2146	243	62	521	7.6	2.5	23
`sort`	2.1	–	9.5	380	–	14800	1740	–	11250	250	–	575	8.0	–	24.5
`sets`	1.6	1.1	22	390	96	2215	3000	380	26093	380	240	4380	2.5	1.5	38

[a] Memory overflow with 118Mo

[b] The model used is not fair for ECLiPSe because of bad propagations on equalities. The solution provided with the ECLiPSe distribution gives results similar to the ones obtained with FaCiLe (4.3s for 256 queens).

[c] No sort constraint is provided by ILOG Solver.

Table 1. Times in seconds on Ultra Sparc for FaCiLe, Ilog Solver 4.3 and ECLiPSe 5.2.

library of the language. The different modules of the library may be linked to produce an interactive toplevel or included in any standalone application.

The first release of FaCiLe has been made available on January 2001. It has been used in our team in a real-size application (7000 decision variables, 140 sorting constraint, 35000 intermediate boolean variables) for air traffic management[2].

We still continue to add features to the library and the short-term plan is the integration of an invariant (à la Localizer) module.

Acknowledgments The authors would like to acknowledge the numerous remarks and suggestions of the reviewers.

Availability FaCiLe sources, examples and documentation are available from:

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

It requires only the OCaml system and runs on any platform supported by the compiler.

References

1. Doug Bagley. The great computer language shootout. Web Page, 2001. `www.bagley.org/~doug/shootout/`.
2. Nicolas Barnier and Pascal Brisset. Slot allocation in air traffic flow management. In *PACLP*, 2000.
3. Chiu Wo Choi, Martin Henz, and Ka Boon Ng. Components for state restoration in tree search. In Toby Walsh, editor, *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, Cyprus, 2001. LNCS.

4. Antonio J. Fernández and Patricia M. Hill. A comparative study of eight constraint programming languages over the boolean and finite domains. *Constraints*, 5(3):275–301, July 2000.
5. Philippe Galinier, Brigitte Jaumard, Rodrigo Morales, and Gilles Pesant. A constraint-based approach to the golomb ruler problem. In *CPAIOR'01*, pages 321–334, April 2001. `www.icparc.ic.ac.uk/cpAIOR01`.
6. 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.
7. Martin Henz and Gert Smolka. Design of a finite domain constraint programming library for ML. `www.comp.nus.edu.sg/ henz/projects/rooms`, 1999.
8. P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
9. 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.
10. François Laburthe. Choco: implementing a CP kernel. In *TRICS Workshop - CP2000*, 2000.
11. 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.
12. Xavier Leroy. The Objective Caml System: User's and reference manual (`http://caml.inria.fr`), 2000.
13. David McClain. A comparison of programming languages for scientific processing. `www.azstarnet.com/~dmcclain/LanguageStudy.html`, January 1999.
14. Laurent Michel and Pascal Van Hentenryck. Modeler++: A modeling layer for constraint programming libraries. `www.cs.brown.edu/people/pvh`, 2001.
15. Greg Morrisett and John Reppy. The third annual ICFP programming contest. Web Page, September 2000. `www.cs.cornell.edu/icfp/`.
16. Jean-François Puget. PECOS: A high-level constraint programming language. In *SPICIS'92*, Singapore, 1992. LNCS.
17. Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.
18. Jeffrey Mark Siskind and David Allen McAllester. Nondeterministic lisp as a substrate for constraint logic programming. In Richard Fikes and Wendy Lehnert, editors, *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Menlo Park, California, 1993. AAAI Press.
19. Barbara Smith. Reducing symmetry in a combinatorial design problem. In *CPAIOR'01*, pages 351–359, April 2001. `www.icparc.ic.ac.uk/cpAIOR01`.
20. Gert Smolka. Concurrent constraint programming based on functional programming. In Chris Hankin, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, vol. 1381, pages 1–11, Lisbon, Portugal, 1998. Springer-Verlag.
21. Solver. ILOG Solver 4.4 user's manual (`www.ilog.fr`), 1999.
22. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, February 1995.
23. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.
24. Pascal Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.