Designing a Generic Graph Library using ML Functors

Sylvain Conchon[1], Jean-Christophe Filliâtre[1], and Julien Signoles[2*]

[1] LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Futurs, ProVal, Orsay F-91893
{conchon,filliatr}@lri.fr
[2] CEA-LIST, Laboratoire Sûreté des Logiciels
Julien.Signoles@cea.fr

Abstract

This paper details the design and implementation of OCAMLGRAPH, a highly generic graph library for the programming language OCAML. This library features a large set of graph data structures—directed or undirected, with or without labels on vertices and edges, as persistent or mutable data structures, etc.—and a large set of graph algorithms. Algorithms are written independently from graph data structures, which allows combining user data structure (resp. algorithm) with OCAMLGRAPH algorithm (resp. data structure). Genericity is obtained through massive use of the OCAML module system and its functions, the so-called *functors*.

1 INTRODUCTION

Finding a graph library for one's favorite programming language is usually easy. But applying the provided algorithms to one's own graph data structure or building undirected persistent graphs with vertices and edges labeled with data other than integers is likely to be more difficult. Figure 1 quickly compares several graph libraries according to the following criteria: number of graph data structures; purely applicative or imperative nature of the structures; and ability to apply the provided algorithms to a user-defined graph data structure. As one can notice, none of these libraries gives full satisfaction. This paper introduces OCAMLGRAPH [1], a highly generic graph library for the programming language OCAML [7], which intends to fulfill all these criteria.

OCAMLGRAPH introduces genericity at two levels. First, OCAMLGRAPH does not provide a single data structure for graphs but many of them, enumerating all possible variations (19 altogether)—directed or undirected graphs, persistent or mutable data structures, user-defined labels on vertices or edges, etc.—under a common interface. Secondly, OCAMLGRAPH provides a large set of graph algorithms that are written independently from the underlying graph data structure. These can then be applied naturally to the data structures provided by OCAMLGRAPH itself and also on user-defined data structures as soon as these implement a minimal set of functionalities.

[*] This work was mostly done while the third author was at LRI.

[1] http://www.lri.fr/~filliatr/ocamlgraph/

	language	graph data structures	persistent / imperative	generic algorithms
GTL [5]	C++	1	I	$\oslash$
LEDA [15]	C++	2	I	$\oslash$
BGL [2]	C++	2	I	$\surd$
JDSL [4]	Java	1	I	$\surd$
FGL [3]	Haskell	1	P	$\surd$
MLRisc [6]	SML	1	I	$\oslash$
Baire [1][2]	OCAML	8	P/I	—

FIGURE 1. Comparison with other graph libraries

Without proper parameterization, such a large set of variants may easily result in unmanageable code. We avoid this pitfall using the OCAML module system [14], which appears to be the tool of choice for this kind of meta-programming. The genericity of OCAMLGRAPH is indeed achieved through a massive use of OCAML functors. On one hand, they are used to avoid code duplication between the many variations of graph data structures, which is mandatory here due to the high number of similar but different implementations. On the other hand, they are used to write graph algorithms independently from the underlying graph data structure, with as much genericity as possible while keeping efficiency in mind.

This paper is organized as follows. Section 2 demonstrates the use of OCAMLGRAPH through an example. Section 3 exposes the design of the common interface for all graph data structures and explains how the code is shared among various implementations. Section 4 describes the algorithms provided in OCAMLGRAPH and how genericity is obtained with respect to the graph data structure. Finally Section 5 presents some benchmarks.

2 MOTIVATING EXAMPLE

To illustrate the use of OCAMLGRAPH, we consider a Sudoku solver based on graph coloring. The idea is to represent the Sudoku grid as an undirected graph with 9×9 vertices, each vertex being connected to all other vertices on the same row, column and 3×3 group. Solving the Sudoku is equivalent to 9-coloring this graph. Figure 2 displays the sketch of a solution to this problem using OCAMLGRAPH [3]. There are four steps in this code:

1. We choose a graph data structure for our Sudoku solver: it is an imperative undirected graph with vertices labeled with pairs of integers (the cells coordinates) and unlabeled edges. In this structure, vertices are also equipped with integer marks, that are used to store the assigned colors.

[2]The Baire library seems to be no longer available from Internet.
[3]Full source code for the Sudoku example is given in appendix A.

```
module G = Imperative.Graph.Abstract
             (struct type t = int × int end)

let g = G.create ()

let () =
  ... add vertices to g with G.add_vertex,
      edges with G.add_edge and initial
      constraints (20 lines of code) ...

module C = Coloring(G)

let () = C.coloring g 9
```

FIGURE 2. A Sudoku solver using OCAMLGRAPH

2. We create the Sudoku grid and fill it with the initial constraints.
3. We obtain a coloring algorithm for our graph data structure.
4. We solve the Sudoku problem by 9-coloring the graph.

This code is almost as efficient as a hand-coded Sudoku solver: on the average, a Sudoku puzzle is solved in 0.2 seconds (on a Pentium IV 2.4 GHz). The remainder of this paper goes into more details about the code above.

3 SIGNATURES AND GRAPH DATA STRUCTURES

Managing many variants of graph data structures without proper parameterization results into unmanageable code. Here we show how we factorized the OCAMLGRAPH implementation to avoid such pitfall. Section 3.1 describes the common sub-signatures shared by all graphs. Section 3.2 details their various implementations.

3.1 Sharing Signatures for All Graphs

All graph data structures share a common sub-signature `G` for observers. Two other signatures distinguish the modifiers for persistent and imperative graphs, respectively.

The common signature `G` includes an abstract type `t` for the graph datatype and two modules `V` and `E` for vertices and edges respectively. The signature for `E` always includes a type `label` which is instantiated by the singleton type `unit` for unlabeled graphs. Modules `V` and `E` both implement the standard comparison and hashing functions so that graph algorithms may easily construct data structures containing vertices and edges. `G` also includes usual observers such as functions to iterate over vertices and edges, which are massively used in graph algorithms. The common signature looks like:

```
module type G = sig
  type t
  module V : sig type t ... end
  module E : sig
    type t
    type label
    val label : t → label
    ...
  end
  val iter_vertex : (V.t → unit) → t → unit
  val iter_succ : (V.t → unit) → t → V.t → unit
  ...
end
```

We distinguish the signature `P` for persistent graphs from the signature `I` for imperative graphs, since the modifiers do not have the same type in both:

```
module type P = sig
  include G
  val empty : t
  val add_vertex : t → V.t → t
  val add_edge : t → V.t → V.t → t
  ...
end
module type I = sig
  include G
  val create : unit → t
  val add_vertex : t → V.t → unit
  val add_edge : t → V.t → V.t → unit
  ...
end
```

3.2 19 Graph Data Structures in 1000 Lines of Code

OCAMLGRAPH provides 19 graph data structures, which include all the possible combinations of the following 4 criteria:

- *directed* or *undirected* graph;
- *labeled* or *unlabeled* edges;
- *persistent* or *imperative* data structure;

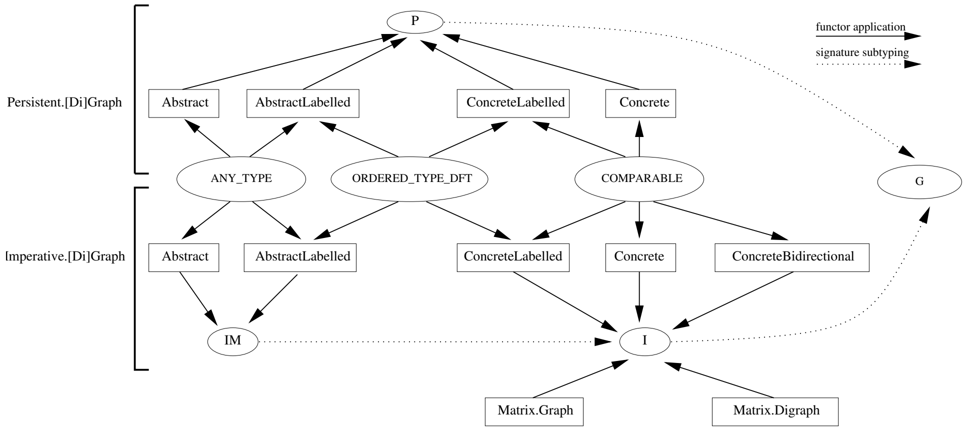

FIGURE 3. OCAMLGRAPH data structures components

- *concrete* or *abstract* type for vertices.

The last point requires some explanations. Vertices are always labeled internally with the value provided by the user. Accessing this value depends on the choice of concrete or abstract vertices. Concrete vertices allow unrestricted access to their value. Abstract vertices hide their value inside an abstract data type. The former allows a more immediate use of the data structure and the latter a more efficient implementation. In particular, imperative graphs with abstract vertices can be equipped with integer mutable marks, which are used in our Sudoku solver.

A functor is provided for each data structure. It is parameterized by user types for vertex labels and possibly edge labels. These functors are displayed in Figure 3 as square boxes mapping signatures of input modules to the signature of the graph module. Of these, 8 functors exist in both directed and undirected versions. Input signatures `ANY_TYPE`, `ORDERED_TYPE_DFT` and `COMPARABLE` define the user types for vertices and edges labels. Output signature `IM` extends signature `I` with mutable marks. Three other implementations complete the set of graph data structures, namely `ConcreteBidirectional` for graphs with an efficient access to predecessors, and `Matrix.(Graph, Digraph)` for graphs implemented as adjacency matrices. For efficiency reasons, these three implementations do not offer the same combination of criteria as the previous ones.

Several functors are used internally to avoid code duplication among the functors presented in Figure 3. For instance, a functor adds labels to unlabeled graphs; another one encapsulates concrete vertices into an abstract data type; etc. Putting it all together, the code size for the 19 graph data structures is about 1000 lines. This is clearly small enough to be easily maintained. In Section 5 we will show that this code is also quite efficient.

The graph data structure for our Sudoku solver is simply an imperative undirected graph with abstract vertices labeled with pairs of integers and unlabeled edges. It is obtained as:

```
module G = Imperative.Graph.Abstract
             (struct type t = int × int end)
```

4 GENERIC ALGORITHMS

This section introduces the second use of functors in OCAMLGRAPH: generic programming of graph algorithms.

4.1 Decoupling Algorithms and Graph Data Structures

As demonstrated in Section 3, our library provides many graph data structures. It makes it necessary to factorize the code for graph algorithms that operate on these structures. Again, functors provide a nice encoding of generic algorithms.

The basic idea when coding an algorithm is to focus only on the required operations that this algorithm imposes on the graph data structure. Then this algorithm can be expressed naturally as a functor parameterized by these operations. These operations usually form a subset of the operations provided by OCAMLGRAPH graph data structures. In a few cases, the algorithm requires specific operations that are independent of the graph data structure. In such a case, the specific operations are provided as an additional functor parameter.

Such a "functorization" of algorithms has two benefits: first, it allows to add quickly new algorithms to the library, without duplicating code for all data structures; secondly, it allows the user to apply an existing algorithm on his own graph data structure.

4.2 Example: Depth-First Traversal

We illustrate the generic programming of graph algorithms on the particular example of depth-first prefix traversal (DFS). To implement DFS, we need to iterate over the graph vertices and over the edges leaving a given vertex. If we do not assume any kind of marks on vertices, we also need to build a data structure to store the visited nodes. We choose a hash table for this purpose and thus we require a hash function and an equality over vertices. Thus the minimal input signature for the DFS functor is as follows:

```
module type G = sig
  type t
  module V : sig
    type t
    val hash : t → int
    val equal : t → t → bool
  end
  val iter_vertex : (V.t → unit) → t → unit
  val iter_succ : (V.t → unit) → t → V.t → unit
end
```

The DFS algorithm is then implemented as a functor with an argument of signature G. The result of functor application is a module providing a single function `dfs` to traverse a given graph while applying a given function on all visited vertices:

```
module Dfs(G : G) :
  sig val dfs : (G.V.t → unit) → G.t → unit end
```

To implement this functor, we first instantiate OCAML's generic hash tables on graph vertices:

```
module Dfs(G : G) = struct
  module H = Hashtbl.Make(G.V)
```

Then we can implement the traversal. The following code uses a hash table `h` to store the vertices already visited and an explicit stack `stack` to store the vertices to be visited (to avoid the possible stack overflow of a recursive implementation). Function `G.iter_vertex` is used to start a DFS on every vertex. The DFS itself is performed in function `loop` using `G.iter_succ`:

```
  let dfs f g =
    let h = H.create 65537 in
    let stack = Stack.create () in
    let push v =
      if not (H.mem h v) then
      begin H.add h v (); Stack.push v stack end
    in
    let loop () =
      while not (Stack.is_empty stack) do
        let v = Stack.pop stack in
        f v;
        G.iter_succ push g v
      done
    in
    G.iter_vertex (fun v → push v; loop ()) g
end
```

Beside this simple algorithm, OCAMLGRAPH provides other kinds of traversals (breadth-first, postfix, etc.) and more efficient implementations when the graph data structure contains mutable marks on vertices.

4.3 Example: Graph Coloring

As a second example, we present a graph coloring algorithm used in our Sudoku solver. For the purpose of our algorithm, we require the presence of `get` and `set` operations on integer marks associated to vertices. We use these marks to store the color assigned to each vertex. We also need iterators over vertices and successors. Thus the minimal signature for a graph data structure used in our graph coloring algorithm is the following:

```
module type GM = sig
  type t
  module V : sig type t ... end
  module Mark : sig
    val get : V.t → int
    val set : V.t → int → unit
  end
  val iter_vertex : (V.t → unit) → t → unit
  val iter_succ : (V.t → unit) → t → V.t → unit
end
```

OCAMLGRAPH already provides implementations for such a signature. This is the case for the graph data structure used in our Sudoku solver. Then the graph coloring algorithm is implemented as the following functor:

```
module Coloring(G : GM) : sig
  val coloring : G.t → int → unit
end
```

It provides a single function `coloring` which colors a given graph with a given number of colors. Some marks may contain initial constraints. To complete our Sudoku solver, we simply need to apply the above functor on our graph module `G`:

```
module C = Coloring(G)
```

If `g` contains the Sudoku graph, and assuming that the initial constraints are set in `g` marks, solving the Sudoku amounts to 9-coloring graph `g`:

```
C.coloring g 9
```

5 BENCHMARKS

Surprisingly, we could not find any standard benchmark for graph libraries. In order to give an idea of OCAMLGRAPH efficiency, we present here the results of a little benchmark of our own. We test four different data structures for undirected graphs with unlabeled edges, that are either persistent (P) or imperative (I) and with either abstract (A) or concrete (C) vertices. In the following, these are referred to as PA, PC, IA and IC, respectively. All tests were performed on a Pentium 4 2.4 GHz.

We first test the efficiency of graph creation and mutation. For that purpose, we build cliques of V vertices (and thus $E = V(V+1)/2$ edges since we include self loops). Then we repeatedly delete all edges and vertices in these graphs. Figure 4 displays the creation and deletion timings in seconds up to $V = 1000$ (that is half a million edges). The speed of creation observed is roughly 100,000 edges per second for imperative graphs. The creation of persistent graphs is slower but within a constant factor (less than 2). Deletion is twice as fast as creation. Regarding

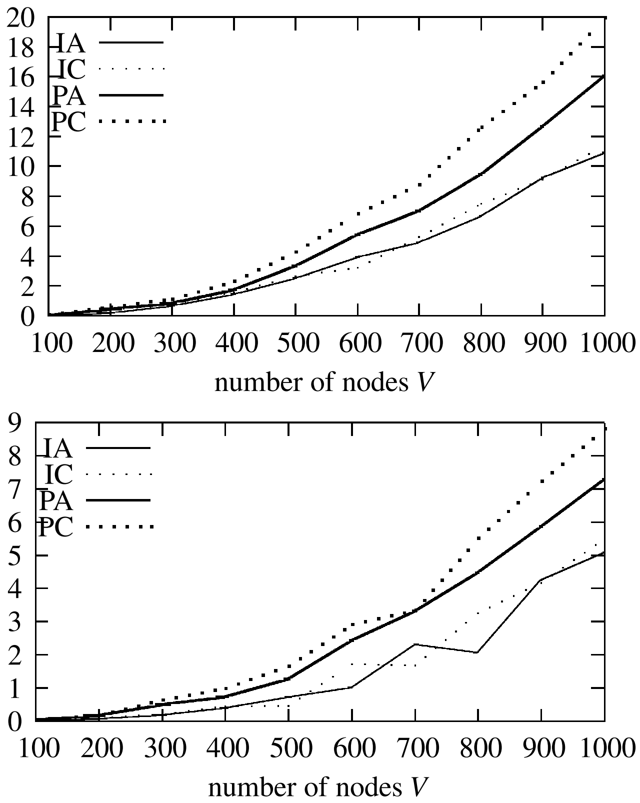

FIGURE 4. Benchmarking creation (top) and deletion (bottom)

memory consumption, all four data structures use approximately 5 machine words (typically 20 bytes) per edge.

Our second benchmark consists in building graphs corresponding to 2D mazes using a percolation algorithm and traversing them using depth-first and breadth-first traversals. Given an integer N, we build a graph with $V = N^2$ vertices and $E = V - 1$ edges. Figure 5 displays the timings in seconds for various values of N up to 600 (i.e. 360,000 vertices). The observed speed is between 500,000 and 1 million traversed edges per second.

We also tested the adjacency matrix-based data structure. Creation and deletion are much faster in that case, and the data structure for a dense graph is usually much more compact (it is implemented using bit vectors). However, the use of this particular implementation is limited to unlabeled imperative graphs with integer vertices. The above benchmarks, on the contrary, do not depend on the nature of vertices and edges types. Thus they are much more representative of OCAMLGRAPH average performances.

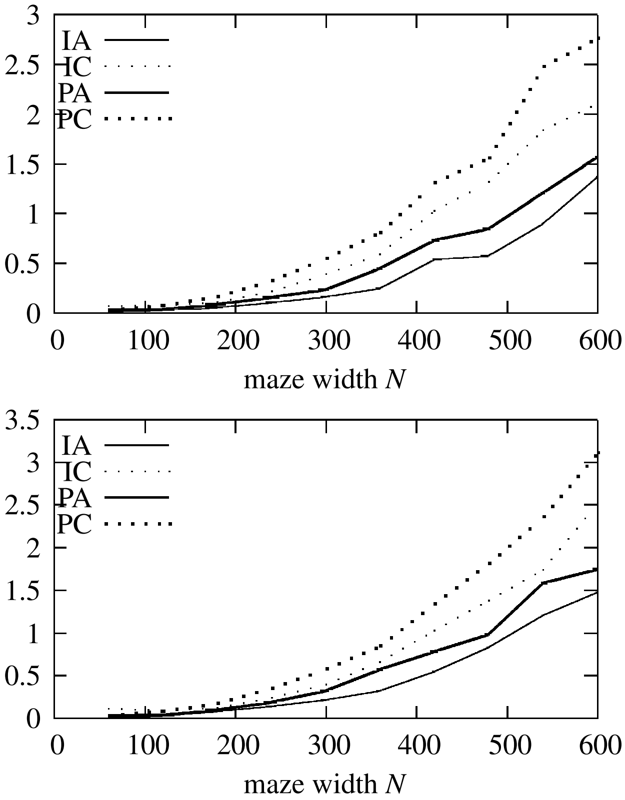

FIGURE 5. Benchmarking DFS (top) and BFS (bottom)

6 CONCLUSION

We presented OCAMLGRAPH, a highly generic graph library for OCAML providing several graph data structures and graph algorithms. Algorithms are written independently from graph data structures, which allows combining user data structure (resp. algorithm) with OCAMLGRAPH algorithm (resp. data structure). To our knowledge, there is no library for any applicative language as generic as OCAMLGRAPH. This genericity is obtained using OCAML module system and especially its functors which allow sharing large pieces of code and provide greater consistency than the mere use of higher-order functions. The same level of genericity could probably be achieved using Haskell's multi-parameter type classes [13]. Regarding imperative languages, graph libraries are rarely as generic and never provide as many different data structures.

Since its first release (Feb. 2004), the number of OCAMLGRAPH users has been increasing steadily and several of them contributed code to the library. Some of them provided new graph data structures (such as `ConcreteBidirectional`) and others new algorithms (e.g. minimal separators). It clearly shows the benefits of a generic library where data structures and algorithms are separated.

REFERENCES

[1] Baire. `http://www.edite-de-paris.com.fr/~fpons/Caml/Baire/`.

[2] BGL - The Boost Graph Library. `http://www.boost.org/libs/graph/doc/`.

[3] FGL - A Functional Graph Library. `http://web.engr.oregonstate.edu/~erwig/fgl/`.

[4] The Data Structures Library in Java. `http://www.cs.brown.edu/cgc/jdsl/`.

[5] The Graph Template Library. `http://infosun.fmi.uni-passau.de/GTL/`.

[6] The MLRISC System. `http://cs1.cs.nyu.edu/leunga/www/MLRISC/Doc/html/INTRO.html`.

[7] The Objective Caml language. `http://caml.inria.fr/`.

[8] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[9] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[10] Jean-Christophe Filliâtre. Backtracking Iterators. Research Report 1428, LRI, Université Paris-Sud, January 2006. `http://www.lri.fr/~filliatr/ftp/publis/enum-rr.ps.gz`.

[11] L. R. Jr. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, pages 99–404, 1956.

[12] Andrew V. Goldberg. *Efficient graph algorithms for sequential and parallel computers*. PhD thesis, MIT, Cambridge, Massachusetts, January 1987.

[13] Oleg Kiselyov. Applicative translucent functors in Haskell, 2004. At `http://www.haskell.org/pipermail/haskell/2004-August/014463.html`.

[14] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[15] Kurt Mehlhorn and Stefan Nher. Leda: a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995.

[16] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[17] Norman Ramsey. ML Module Mania: A Type-Safe Separately Compiled, Extensible Interpreter. In *ACM SIGPLAN Workshop on ML*, 2005.

A SIMPLE SUDOKU SOLVER USING OCAMLGRAPH

Below is the full listing for a Sudoku solver using OCAMLGRAPH, as described in this paper. This program reads the Sudoku problem on standard input and prints the solution on standard output.

```
open Graph

(* We use undirected graphs with nodes containing
   a pair of integers (the cell coordinates in
   0..8 x 0..8). *)
module G = Imperative.Graph.Abstract
             (struct type t = int * int end)

(* The Sudoku grid = a graph with 9x9 nodes *)
let g = G.create ()

(* We create the 9x9 nodes, add them to the graph
   and keep them in a matrix for later access *)
let nodes =
  let new_node i j =
    let v = G.V.create (i, j) in G.add_vertex g v; v
  in
  Array.init 9 (fun i -> Array.init 9 (new_node i))

let node i j = nodes.(i).(j)

(* We add the edges: two nodes are connected whenever
   they can't have the same value *)
let () =
  for i = 0 to 8 do for j = 0 to 8 do
    for k = 0 to 8 do
      if k <> i then G.add_edge g (node i j) (node k j);
      if k <> j then G.add_edge g (node i j) (node i k);
    done;
    let gi = 3 * (i / 3) and gj = 3 * (j / 3) in
    for di = 0 to 2 do for dj = 0 to 2 do
      let i' = gi + di and j' = gj + dj in
      if i' <> i || j' <> j then
        G.add_edge g (node i j) (node i' j')
    done done
  done done

(* We read the initial constraints from standard input *)
let () =
  for i = 0 to 8 do
    let s = read_line () in
    for j = 0 to 8 do match s.[j] with
      | '1'..'9' as ch ->
        G.Mark.set (node i j) (Char.code ch - Char.code '0')
      | _ -> ()
    done
  done

(* We solve the Sudoku by 9-coloring the graph g *)
```

```
module C = Coloring.Mark(G)
let () = C.coloring g 9

(* We display the solution *)
let () =
  for i = 0 to 8 do
    for j = 0 to 8 do
      Format.printf "%d" (G.Mark.get (node i j))
    done;
    Format.printf "\n";
  done;
  Format.printf "@?"
```