Designing a Generic Graph Library using ML Functors

Submitted for blind review

Abstract

This article details the design and implementation of OCAMLGRAPH, a 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 written independently of the graph data structure. Such a genericity is obtained through a massive use of OCAML module system and of its functions, the so-called *functors*.

Categories and Subject Descriptors D.2 [*Software Engineering*]: Software Architectures; Design; Reusable Software

Keywords Graph library, Generic programming, Functors

1. Introduction

Finding a graph library for one's favorite programming language is usually easy. But using the provided algorithms on one's own graph data structure or building undirected persistent graphs with vertices and edges labeled with something else than integers is likely to be more difficult.

This article introduces OCAMLGRAPH [1], a generic graph library for the programming language OCAML [7]. Beside the mere presentation of this library, this article demonstrates how OCAML has been used to introduce genericity at two levels in this library. First, OCAMLGRAPH does not provide a single data structure for graphs but many of them, enumerating all possible variations—directed or undirected graphs, persistent or mutable data structures, user-defined labels on vertices or edges, etc.—under a common interface. Second, OCAMLGRAPH provides a large set of graph algorithms that are written independently of the underlying graph data structure. Then they can be applied on the data structures provided by OCAMLGRAPH itself but also on user-defined data structures as soon as they implement a minimal set of functionalities.

The genericity of OCAMLGRAPH, on both levels of data structures and algorithms, is realized through a massive use of OCAML module system [13]. In the first case, it is used to avoid code duplication between the many variations of graph data structures, which is mandatory here seen the high number (19) of similar but all different implementations. In the second case, it is used to write the graph algorithms independently of the underlying graph data structure, with as much genericity as possible but also with an efficiency concern in mind. In both cases, the technical solution comes from the functions of OCAML module system, the so-called *functors* in the functional programming jargon (that should not be confused with C++ functors). This article aims at explaining these two different uses of functors and thus is more a software engineering article than a graph library description.

This article is organized as follows. Section 2 briefly introduces the OCAML module system. Then Section 3 exposes the design of the common interface for all graph data structures and explains how the code is shared among the various implementations. Section 4 describes the algorithms provided in OCAMLGRAPH and how the genericity with respect to the graph data structure is obtained. Section 5 illustrates the use of OCAMLGRAPH on two typical situations. Finally Section 6 presents some benchmarks and Section 7 compares OCAMLGRAPH with several existing graph libraries.

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

[copyright notice will appear here]

2. OCAML Module System

This section quickly describes the OCAML module system. Any reader familiar with OCAML can safely skip this section.

The module system of OCAML is a language by itself, on top of the core OCAML language, which only fulfills software engineering purposes: separate compilation, names space structuring, encapsulation and genericity. This language appears to be independent of the core language [13] and may be unfolded statically. It is a strongly typed higher-order functional language. Its terms are called *modules* or *structures*. They are the basic blocks in OCAML programs, that package together types, values, exceptions and submodules.

2.1 Structures

Modules are introduced using the `struct...end` construct and the (optional) `module` binding is used to give them a name. Outside a module, its components can be referred to using the *dot notation*: `M.c` denotes the component `c` defined in the module `M`.

For instance, a module packaging together a type for a graph data structure and some basic operations can be implemented in the following way:

```
module Graph = struct
  type label = int
  type t = (int * label) list array
  let create n = Array.create n []
  let add_edge g v1 v2 l = g.(v1) <- (v2,l)::g.(v1)
  let iter_succ g f v = List.iter f g.(v)
end
```

The type `Graph.t` defines a naive graph data structure using adjacency lists with edges labeled with integers: a graph is an array (indexed by integers representing vertices) whose elements are lists of pairs of integers and labels (declared as an alias for the type `int`).

2.2 Signatures

The type of a module is called a *signature* or an *interface*[2] and can be used to hide some components or the definition of a type (then called an *abstract data type*). Signatures are defined using the `sig...end` construct and the (optional) `module type` binding is used to give them a name. Constants and functions are declared via the keyword `val` and types via the keyword `type`.

For instance, a possible signature for the `Graph` module above, that hides the graph representation and the type of labels, could be the following:

```
module type GRAPH = sig
  type label
  type t
  val create : int -> t
  val add_edge : t -> int -> int -> label -> unit
  val iter_succ :
    t -> (int * label -> unit) -> int -> unit
end
```

Restricting a structure by a signature results in another view of the structure. This is done as follows:

```
module G' = G : GRAPH
```

Since interfaces and structures are clearly separated, it is possible to have several implementations for the same interface. Conversely, a structure may have several signatures (hiding and restricting more or less components).

2.3 Functors

The functions of the module system are called *functors* and allow us to define modules parameterized by other modules. Then they can be applied one or several times to particular modules with the expected signature. The benefits of functors in software engineering are appreciated as soon as one has to parameterize a *set* of types and functions by another *set* of types and functions, in a sound way[3]. For instance, to implement Dijkstra's shortest path algorithm for any graph implementation where edges are labeled with integers, one can write a functor looking like:

```
module type S = sig
  type label
  type t
  val iter_succ :
    t -> (int * label -> unit) -> int -> unit
end

module Dijkstra (G : S with type label = int) =
  struct
    let dijkstra g v1 v2 = (* ... *)
  end
```

The `with type` annotation is used here to unify the abstract type `label` from the signature `S` with the type `int`. One may also notice that the signature `S` required for the functor's argument only contains what is necessary to implement the algorithm. However, we can apply the functor to any module whose signature contains *at least* `S` i.e. is a *subtype* of `S`.

Functors are also first-class values, i.e they can be passed as arguments to other functors. Finally, it is possible to aggregate signatures or modules using the `include` construct which can be naively seen as a textual inclusion.

[2] with the same meaning as in MODULA but not as in JAVA

[3] See for instance Norman Ramsey's *ML Module Mania* [16] as an example of a massive use of ML functors.

3. Graph Data Structures

OCAMLGRAPH does not provide a single data structure for graphs but a set of functors to build customized data structures. More precisely, the user must answer the following questions:

- *What are the graph's vertices?* The user must define the type of the values stored in each vertex. He also has to choose between *concrete* vertices (where vertices and their values are identical) and *abstract* vertices (where the values are hidden inside an abstract data type for vertices). In the latter case, it allows a more efficient implementation of some operations.
- *Are the graphs directed or not?*
- *Are the edges labeled or not?* When they are labeled, the user must define the type of these labels.
- *Are the graphs imperative or persistent?* An imperative graph is a mutable data structure where modifications are performed in-place, while a persistent graph is an immutable data structure. In the latter case, operations such as adding or removing edges and vertices are still available but are returning new graphs[4].

Altogether, not less than 16 different data structures must be implemented to provide all possible combinations. They are provided as functors, since they are parameterized by user types. These 16 functors are displayed Figure 1 as square boxes mapping signatures of input modules to signatures of output modules. For instance, a persistent directed graph with abstract vertices and labeled edges is obtained by applying the functor

`Persistent.Digraph.AbstractLabelled`

to a module defining the type for the vertices values (with signature `ANY_TYPE` defined as `sig type t end`) and to a module defining the type for the edges labels (with signature `ORDERED_TYPE_DFT` that declares a type `t`, a total order over `t` and a default value of type `t`).

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.

The remaining of this section details the signatures and the modules displayed Figure 1. More precisely, Section 3.1 shows how the interfaces for all the data structures are unified in order to ease the access to the library. Then Section 3.2 explains how the code is shared between the various implementations.

3.1 Interfaces

The various graph data structures interfaces are gathered in the module `Sig`. There are three main signatures: a common signature `G` for all data structures, a signature `P` for persistent data structures and a signature `I` for imperative data structures. The common interface `G` contains all access operations. The interfaces `P` and `I` extend `G` by adding creation and modification operations, obviously different for persistent and imperative data structures.

The common interface `G` introduces an abstract data type `t` for graphs:

```
module type G = sig
  type t                         (* the type of graphs *)
```

The type of vertices is also an abstract type[5]. Since we do not know yet whether the vertices will be labeled or not, we also introduce a type for vertices labels. (One could have considered a polymorphic

[4] Implementing persistent (sometimes also called *functional*) data structures can be done efficiently; see for instance Okasaki's book [15].

[5] The type of vertices is abstract in this generic interface but a particular implementation may declare a manifest type.

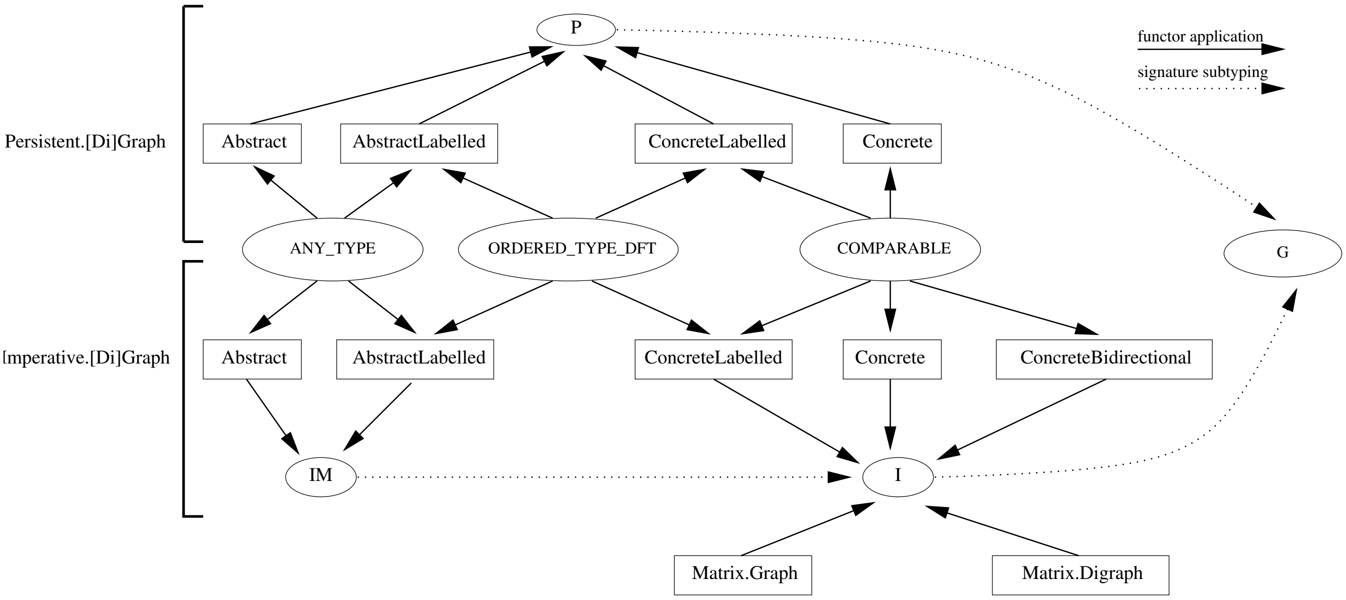

Figure 1. OCAMLGRAPH's data structures components

type for vertices but polymorphism does not marry nicely with functors.) Finally, most graph algorithms require to store vertices in data structures such as sets or dictionaries and thus it is convenient to have the type of vertices equipped with comparison and hashing functions. We gather all these types and functions within a sub-module `V`:

```
module V : sig
  type t              (* the type of vertices *)
  type label                  (* vertices labels *)
  val create : label -> t
  val label : t -> label
  val compare : t -> t -> int
  val hash : t -> int
  val equal : t -> t -> bool
end
```

Introducing a sub-module has several advantages: first, it avoids polluting the name space; second, it allows this sub-module to be directly used as an argument of functors from OCAML's standard library. Yet we introduce an alias for the type `V.t` for a greater clarity in the following:

```
type vertex = V.t
```

For graphs where vertices are unlabeled, we will identify the types `V.t` and `V.label` and we will implement the operations `V.create` and `V.label` as the identity.

Similarly, we introduce a sub-module `E` for edges. Again, we assume that the edges may be labeled and an edge is thus built from two vertices and a label. Finally, we equip the edges type with a comparison function as we did for vertices.

```
module E : sig
  type t                 (* the type of edges *)
  type label                  (* edges labels *)
  val create : vertex -> label -> vertex -> t
  val src : t -> vertex
  val dst : t -> vertex
  val label : t -> label
  val compare : t -> t -> int
end
```

```
type edge = E.t
```

For unlabeled edges, we will implement the type `E.t` as `V.t * V.t` and we will ignore the type `E.label` (implementing it with the `unit` type for instance).

Then we introduce a large set of test, access and iteration functions over vertices and edges, whose meaning is immediate:

```
val is_directed : bool
val nb_vertex : t -> int
val out_degree : t -> vertex -> int
val iter_vertex : (vertex -> unit) -> t -> unit
(* ... *)
```

The central part of the interface is the set of functions giving access to the graph structure. One can find for instance a function returning the successors of a vertex as a list of vertices:

```
val succ : t -> vertex -> vertex list
(* ... *)
```

In practice, however, it is more useful to iterate a given function over the successors (or the predecessors), to avoid building a list that would be immediately deconstructed. This set of iterators completes the signature `G`:

```
  val iter_succ :
    (vertex -> unit) -> t -> vertex -> unit
  val fold_succ :
    (vertex -> 'a -> 'a) -> t -> vertex -> 'a -> 'a
  (* ... *)
end
```

As demonstrated in Section 4, implementing a graph algorithm usually requires a small subset of these operations. Using signatures sub-typing, any implementation of the signature `G` will also be an adequate signature for our algorithms.

The interface `P` for persistent graphs then extends the interface `G` with a set of creation functions:

```
module type P = sig
  include G
  val empty : t
```

```
  val add_vertex : t -> vertex -> t
  val remove_vertex : t -> vertex -> t
  val add_edge : t -> vertex -> vertex -> t
  (* ... *)
end
```

Similarly, the interface I for imperative graphs extends G with creation and modification functions:

```
module type I = sig
  include G
  val create : unit -> t
  val copy : t -> t
  val add_vertex : t -> vertex -> unit
  val add_edge : t -> vertex -> vertex -> unit
  (* ... *)
end
```

One can notice the copy function which has no counterpart in the interface P.

In Section 4.4 we will show how to give a common interface to graphs creation, whatever their persistent or imperative nature is, in order to share code in some algorithms that need to build graphs. But it would have been unfortunate to do it here, since the mutability of the graph must appear clearly to the user when looking at the signature.

Finally, the module Sig introduces a fourth (and last) signature for graphs, IM, for imperative data structures where vertices are equipped with integers *marks* which can be modified in-place:

```
module type IM = sig
  include I
  module Mark : sig
    val clear : t -> unit  (* set all marks to 0 *)
    val get : vertex -> int
    val set : vertex -> int -> unit
  end
end
```

Such a signature is useful when one wants to provide an efficient implementation of an algorithm based on a specific marking of the vertices. Moreover, such marks can then be exploited by the user once the algorithm's run is completed, since they are exposed in the signature. The typical example is a depth-first traversal.

One can notice that the directed nature of the graph does appear nowhere in the signature. Only the *implementations* will set it.

3.2 Code Sharing

The implementations of the various data structures for graphs are gathered in the two modules Persistent and Imperative. Each of these two modules is divided into two sub-modules: Graph for undirected graphs and Digraph for directed ones. Each of these sub-modules contains four functors, corresponding to concrete or abstract data types[6] with labeled or unlabeled edges (vertices are always labeled). These functors are parameterized by a module V giving the type of the vertices and, when meaningful, by a module E giving the type of the edges. For instance, the functor Persistent.Digraph.Abstract implements directed persistent graphs, implemented as an abstract data type and with no labels on edges. Its signature is the following:

```
  module Abstract(V: sig type t end) :
    Sig.P with type V.label = V.t
```

In the remaining of this section, we show how functors are used to share as much code as possible among these 16 data structures.

[6] An abstract graph hides the representation of vertices and edges, allowing an optimized implementation of some operations.

Note the code presented here is not useful (and actually not visible) for the user of the library.

The module Per_imp, which is internal to OCAMLGRAPH, gathers a few functors implementing operations that are common to persistent and imperative graphs. Each functor has a dedicated task, such as adding labels on edges (functor Labelled), makes the graph abstract (functor Make_Abstract) or adding the operations on predecessors (functor Pred). Then implementing a given graph data structure is a matter of assembling the various functors as in a construction game and adding the few operations that could not be factorized.

Abstracting the persistent or imperative nature. In our library, graphs are represented as dictionaries mapping each vertex to the set of its successors (with the corresponding edge's label if any). These dictionaries are either persistent or imperative. In OCAML, it results in operations with different types. Indeed, adding a mapping into an hash table does not return any value but adding it into a persistent map (an AVL for instance) returns a new map. Yet it is possible to give these persistent and imperative maps a common signature using an explicit 'a return type for the returned value:

```
module type HM = sig
  type 'a return
          (* return type for [add] and [remove] *)
  type 'a t
  type key
  val add: key -> 'a -> 'a t -> 'a return
  val mem: key -> 'a t -> bool
  val find: key -> 'a t -> 'a
  (* ... *)
end
```

It is then possible to get a persistent implementation of HM using OCAML's Map module by instantiating the type 'a return with the type of persistent maps:

```
module Make_Map(X: (* ... *) ) = struct
  include Map.Make(X)
  type 'a return = 'a t
  (* ... *)
end
```

Beside, it is also possible to get an imperative implementation using OCAML's Hashtbl module by instantiating the type 'a return with unit:

```
module Make_Hashtbl(X: (* ... *) ) = struct
  include Hashtbl.Make(X)
  type 'a return = unit
  let add k v h = replace h k v
  (* ... *)
end
```

Now, using an implementation of HM together with an implementation of finite sets, it is possible to write a functor implementing the operations which are common to all graphs:

```
module Minimal(S: Set.S)(HM: HM) = struct
  let mem_vertex g v =
    HM.mem v g
  let unsafe_add_edge g v1 v2 =
    HM.add v1 (S.add v2 (HM.find v1 g)) g
  (* ... *)
end
```

Functors as building blocks. Operations over predecessors are always implemented the same way. They are provided by the Pred functor:

```
module Pred(S: (* ... *) ) = struct
  let fold_pred f g v = (* ... *)
  let pred g v =
    fold_pred (fun v l -> v :: l) g v []
  (* ... *)
end
```

Similarly, two functors `Labeled` and `Unlabeled` introduce all operations that depend on the presence or absence of labels on edges, such as iterating over the edges, over the successors of some vertex, etc.

The abstraction of a graph data structure is an operation that encapsulates the type of vertices given by the user in some more efficient data structure (typically a record with a unique integer). When this is done, some operations can be optimized, such as counting the number of vertices in the graph, since the user does not have a direct access to the vertices anymore. The functor `Make_Abstract` realizes such an optimization:

```
module Make_Abstract(G: Sig.G) = struct
  module I = struct
    type t = { edges: G.t; mutable size: int }
    let iter_vertex f g = G.iter_vertex f g.edges
    (* ... *)
  end
  include I
  include Pred(I)
  let nb_vertex g = g.size       (* optimization *)
  (* ... *)
end
```

Finally, it is possible to implement the concrete version of the module `V` and the corresponding association table, given a user data type for the vertices, using a higher-order functor:

```
module ConcreteVertex
  (F: functor(X: (* ... *) ) ->
      HM with type key = X.t)
  (V: (* ... *) ) =
struct
  module V = struct
    include V
    type label = t
    let label v = v
    let create v = v
  end
  module HM = F(V)
end
```

Assembling the blocks. Using all these functors, it is now possible to implement in a generic and easy way the major part of the modules `Digraph` and `Graph` as the following higher-order functor:

```
module Make(F: functor(X: (* ... *) ) ->
               HM with type key = X.t) =
struct
  module Digraph = struct
    module Concrete(V: (* ... *) ) = struct
      include ConcreteVertex(F)(V)
                         (* vertices are concrete *)
      include Unlabeled(V)(HM)
                           (* edges are unlabeled *)
      include Minimal(S)(HM)
                             (* common operations *)
    end
    module ConcreteLabelled = (* ... *)
    module Abstract = (* ... *)
    module AbstractLabelled = (* ... *)
    end
  end
  module Graph = struct (* ... *) end
end
```

The persistent and imperative implementations are then obtained by applying this functor respectively to `Make_Map` and `Make_Hashtbl`:

```
module P = Make(Make_Map)
module I = Make(Make_Hashtbl)
```

Finally, to get the functors provided to the user, we simply need to add the operations that could not be factorized:

```
module Digraph = struct
  module Concrete(V: (* ... *) ) = struct
    include P.Digraph.Concrete(V)
    let add_vertex g v1 v2 =
      if HM.mem v g then g else add_vertex g v
    (* ... *)
  end
  (* ... *)
end
```

Putting all together, the code for the 19 graph data structures amounts to 1059 lines. This is clearly small enough to be easily maintained. In Section 6 we will show that this code is also quite efficient.

4. Algorithms

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

4.1 Generic Programming

As demonstrated in the previous section, our library provides many graph data structures. Therefore, it is necessary to figure out how to implement graph algorithms without duplicating code for each data structure. Exactly as functors helped factorizing code in the previous section, they provide a nice way to code the algorithms in a generic way.

The basic idea is to code an algorithm without worrying about the underlying graph data structure but focusing only on the required operations over this data structure. Then the algorithm is naturally a functor parameterized by these operations. Such operations usually make a subset of the operations provided by OCAMLGRAPH's own data structures, but some algorithms may require specific operations, independent of the graph data structure. In such a case, they will be provided as an additional functor parameter.

Such a "functorization" of algorithms has two benefits: first, it allows to quickly add a new algorithm to the library, without code duplication for all data structures; second, it allows the user to apply an algorithm on its own graph data structure, which is an original feature of OCAMLGRAPH w.r.t. other graph libraries.

4.2 Example: Flow Algorithms

We illustrate the generic implementation of algorithms in OCAMLGRAPH through two algorithms to compute the maximal flow of a network[7], namely the Ford-Fulkerson [11] and Goldberg [12] algorithms. They appear as two functors with the following signatures:

```
module Ford_Fulkerson
  (G: G_FORD_FULKERSON)
  (F: FLOW with type label = G.E.label) :
```

[7] that is a directed graph with two distinguished vertices, a source and a terminal.

```
sig
  val maxflow :
    G.t -> G.V.t -> G.V.t -> (G.E.t -> F.t) * F.t
end

module Goldberg
  (G: G_GOLDBERG)
  (F: FLOW with type label = G.E.label) :
sig
  val maxflow :
    G.t -> G.V.t -> G.V.t -> (G.E.t -> F.t) * F.t
end
```

Each functor body implements a `maxflow` function expecting a network, a source vertex and a terminal vertex, and returning a pair `(f, delta)` where `f` maps each edge of the network to its new flow and where `delta` is the difference between the computed maximal flow and the initial flow of the network.

The signature of the first parameter `G` of these two functors describes the minimal set of graph operations required for the implementation of `maxflow`. Since the two algorithms are different, the signatures `G_FORD_FULKERSON` and `G_GOLDBERG` are thus different. For instance, Goldberg algorithm requires to iterates over the vertices and edges of the graph, whereas Ford-Fulkerson algorithm only requires to iterate over the successors and predecessors of a vertex. This appears clearly in the signatures:

```
module type G_GOLDBERG = sig
  type t
  module V : sig type t (* ... *) end
  module E : sig type t (* ... *) end
  val iter_vertex : (V.t -> unit) -> t -> unit
  val iter_edges_e : (E.t -> unit) -> t -> unit
  (* ... *)
end

module type G_FORD_FULKERSON = sig
  type t
  module V : sig type t (* ... *) end
  module E : sig type t (* ... *) end
  val iter_succ_e :
    (E.t -> unit) -> t -> V.t -> unit
  val iter_pred_e :
    (E.t -> unit) -> t -> V.t -> unit
end
```

The signature of the second parameter `F` is common to both algorithms. It gathers all operations to manipulate the labels on edges as flows:

```
module type FLOW = sig
  type label            (* type of edges labels *)
  type t                        (* type of flow *)
  val flow : label -> t         (* initial flow *)
  val max_capacity : label -> t (* max. capacity *)
  val min_capacity : label -> t (* min. capacity *)
  val add : t -> t -> t             (* addition *)
  val sub : t -> t -> t          (* subtraction *)
  val zero : t               (* neutral element *)
  val compare : t -> t -> int      (* comparison *)
end
```

This signature `FLOW` is easily implemented, whatever the data type is. For instance, a possible implementation for a graph where edges are labeled with integers representing the maximal capacities of edges and with an initial empty flow would be the following:

```
module Flow = struct
  type label = int
  type t = int
  let max_capacity x = x
  let min_capacity _ = 0
  let flow _ = 0
  let add = (+)
  let sub = (-)
  let compare = compare
  let zero = 0
end
```

4.3 Example: Graph Traversal

We give here another example of generic programming with the classical depth-first and breadth-first traversals. To traverse a graph, we need very few operations. Basically, we need to iterate over the successors of a vertex. To reach all the connected components of the graph, we also need to iterate over all vertices. Finally, we have to mark the visited vertices. Regarding this last point, the simplest way is to use a hash table (when no particular assumption can be made about the graph data structure). To build such a hash table, we simply need the type of vertices to be equipped with suitable `hash` and `equal` functions. Therefore, the minimal expected signature for traversal algorithms is the following:

```
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
```

Then traversal algorithms can be provided with a signature like:

```
module Dfs(G : G) :
  sig val iter : (G.V.t -> unit) -> G.t -> unit end
```

In practice, however, several variants of each traversal are provided: traversal of a single connected component, application of the visitor function after the successors are visited (instead of before), etc. The depth-first traversal functor also provides a cycle detection function. Finally, there is a functor specialized for graphs equipped with marks (see signature `IM` in Section 3.1), since such marks can be directly used to mark the visited vertices (and for further use by the library client).

Graph traversals are also provided as *cursors*, i.e. step-by-step iterators similar to those used in object-oriented programming [8] (e.g. the JAVA `Iterator` interface). Such a cursor is given the following signature:

```
module Dfs(G : G) : sig
  type iterator
  val start : G.t -> iterator
  val step : iterator -> iterator
  val get : iterator -> G.V.t
end
```

The abstract type `iterator` represents a state of the iterator, or equivalently a position in the ongoing iteration. We get the starting point with the `start` function. Then we can advance in the iteration using the `step` function and we can obtain the currently visited vertex using the `get` function. These two functions raise the `Exit` exception whenever the iteration is over. There is a huge difference w.r.t. usual cursors in imperative programming, however: the data type `iterator` is persistent. It means that the iterator is not modified when the step function is called, but a new iterator is returned instead. Therefore it is easy to use such iterators in *backtracking* algorithms that may need to come back to previous positions in an iteration [10].

Here is for instance a simple implementation of a k-coloring algorithm based on a breadth-first traversal implemented as a persistent cursor. We assume that the graph g contains integer marks that we use to indicate the color (the value 0 meaning that there is no assigned color yet).

```
let coloring g k =
  Mark.clear g;
  let try_color v i =
    iter_succ
      (fun w ->
        if Mark.get w = i then raise NoColor)
      g v;
    Mark.set v i
  in
  let rec iterate iter =
    let v = Bfs.get iter in
    for i = 1 to k do
      try try_color v i; iterate (Bfs.step iter);
          assert false
      with NoColor -> ()
    done;
    Mark.set v 0; raise NoColor
  in
  try iterate (Bfs.start g); assert false
  with Exit -> ()
```

The benefits of the persistent cursor are twofold: first, we could use an existing traversal function without having to mix its code with the coloring algorithm itself; second, the persistent nature of the iterator avoids an explicit backtracking in the above code.

4.4 Building Graphs

In Section 3.1, we have shown that persistent and imperative graphs have creation functions with different signatures. However, as we have written algorithms in a generic way, we may want to build graphs in a generic way, that is independently of the underlying data structure. For instance, we may want to implement graph operations (such as union, transitive closure, etc.) or to build some classic graphs (such as the full graph with n vertices, the de Bruijn graph of order n, etc.) or even random graphs. In all these cases, the persistent or imperative nature of the graph is not really significant but the signature difference disallows genericity.

To solve this issue, we introduce a module `Builder`. It defines a common interface for graphs building:

```
module type S = sig
  module G : Sig.G
  val empty : unit -> G.t
  val copy : G.t -> G.t
  val add_vertex : G.t -> G.V.t -> G.t
  val add_edge : G.t -> G.V.t -> G.V.t -> G.t
  val add_edge_e : G.t -> G.E.t -> G.t
end
```

It is immediate to realize such a signature for persistent or imperative graphs:

```
module P(G : Sig.P) : S with module G = G
module I(G : Sig.I) : S with module G = G
```

It is important to notice that for imperative graphs the values returned by the functions `add_vertex`, `add_edge` and `add_edge_e` are meaningless.

Therefore, it is easy to write a generic algorithm that builds graphs. First we write a highly generic version as a functor taking a module of signature `Builder.S` as argument:

```
module Make(B : Builder.S) = struct ... end
```

and then we can trivially provide two variants of this functor for both persistent and imperative graphs, with the following two lines:

```
module P(G : Sig.P) = Make(Builder.P(G))
module I(G : Sig.I) = Make(Builder.I(G))
```

Thus the use of the module `Builder` is entirely hidden from the user point of view.

5. Using OCAMLGRAPH

This section demonstrates how to use OCAMLGRAPH's functors on two typical situations: first, a direct use of OCAMLGRAPH's data structures and algorithms to solve a problem; second, a more complex situation where the user applies an OCAMLGRAPH's algorithm on his own graph data structure.

5.1 Sudoku

This section illustrates the use of OCAMLGRAPH to solve the Sudoku puzzle using graph coloring. This idea is to represent the Sudoku grid as an undirected graph with 9×9 vertices, each vertex being connected to the other vertices on the same row, column or 3×3 group. Then solving the Sudoku is equivalent to 9-coloring this graph.

First, we create a new graph data structure where vertices are labeled with pairs of integers (the coordinates in the grid, as integers in the range 0..8):

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

We use the `Imperative` module since persistence is not needed here, the `Graph` sub-module to indicate undirected graphs and the `Abstract` functor both for efficiency and simplicity. Indeed, such graphs come with integers marks on the vertices that will be used here to store the colors.

Then we can create the graph itself:

```
let g = G.create ()
```

This graph is initially empty. We need to create and add the 81 vertices. We could do it as simply as:

```
for i = 0 to 8 do for j = 0 to 8 do
  G.add_vertex g (G.V.create (i,j))
done done
```

but in order to access the vertices in the remaining of the code we store them in a matrix:

```
let vert =
  let new_vertex 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_vertex i))
```

Then we add the edges, connecting the vertices on a same row, column or group:

```
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 vert.(i).(j) vert.(k).(j);
    if k <> j then
      G.add_edge g vert.(i).(j) vert.(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 vert.(i).(j) vert.(i').(j')
  done done
done done
```

The initial constraints in the Sudoku puzzle are set using the `G.Mark.set` function.

To get the coloring algorithm for our graph data structure, we simply apply the `Coloring.Mark` functor on `G`:

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

Finally, solving the Sudoku amounts to 9-coloring the graph g:

```
C.coloring g 9
```

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 4 2.4 GHz).

5.2 Topological Sorting

This section illustrates the use of one of OCAMLGRAPH's traversal algorithms to perform some topological sorting on a set of files dependencies. We assume given the following user-defined data structure representing a file with its dependencies:

```
type file =
  { name : string ; mutable deps : file list }
```

In order to get the set of dependencies for a given file, we just have to visit the underlying graph using a depth-first traversal. For that purpose, we can use OCAMLGRAPH's `Graph.Traverse.Dfs` functor (which is similar to the one presented Section 4.3). To do so, we need to build a module that implements the required graph operations. First, we embed the `file` data type in a `Vertex` module:

```
module Vertex = struct
  type t = file
  let compare n1 n2 = compare n1.name n2.name
  let hash n = Hashtbl.hash n.name
  let equal n1 n2 = n1.name = n2.name
end
```

Then viewing a list of files as a graph is easily implemented by the following module:

```
module G = struct
  type t = file list

  module V = Vertex

  let iter_vertex = List.iter
  let fold_vertex = List.fold_right
  let iter_succ f _ v = List.iter f v.deps
  let fold_succ f _ v = List.fold_right f v.deps
end
```

Finally, we obtain the depth-first traversal module as a functor instantiation:

```
module Dfs = Graph.Traverse.Dfs(G)
```

6. Benchmarks

Surprisingly, we could not find any standard benchmark for graph libraries. Thus we present here the results of a little benchmark of our own, mostly to give the reader a rough idea of OCAMLGRAPH efficiency. 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 they 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 full graphs with 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 2 displays the creation and deletion timings in seconds up to $V = 1000$ (that is half a million of edges). The creation speed 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 roughly twice faster than creation. Regarding memory consumption, all four data structures roughly use 5 machine words (typically 20 bytes) per edge.

Our second benchmark consists in building graphs corresponding to 2D mazes, using a percolation algorithm, and then 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 3 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.

7. Related Work

This section details the features of several other graph libraries, in order to make a comparison with OCAMLGRAPH. Our list is not exhaustive and only focuses on programming languages where some kind of generic programming is possible. Figure 4 summarizes our comparison criteria which are mainly based on the genericity of the data structures and algorithms proposed by some other libraries.

JDSL [4] This JAVA library provides implementations and algorithms for various data structures (lists, vectors, priority queues, dictionaries and graphs). Regarding graphs, JDSL introduces several distinct signatures for access and modification to the underlying data structure: one can find *accessors* for constant-time access to graph elements, *iterators* to traverse the graph structure, and *modifiers* to add or suppress vertices and edges. The use of JAVA interfaces makes JDSL algorithms highly generic. However, the library only provides an implementation of graphs as adjacency lists and there is no notion of persistent graph. Moreover, there is no way to distinguish between directed and undirected graphs (edges and arcs may coexist within a same graph). Finally, vertices and edges may be labeled with values in the class `Object` which is the usual but risky and inefficient way to get polymorphism in JAVA (at least before generics were introduced in JAVA 1.5).

MLRISC [6] This generic compiler back-end provides a graph library for SML which contains a unique signature for directed multi-graphs together with a unique imperative implementation. Vertices and edges may be labeled with information of any type. Modifiers and iterators allow respectively to update and traverse the graph data structure. Combinators are also provided to build different *views* of a single graph. One may for instance build a view where arcs are reversed, where vertices are renamed or where vertices are gathered in subsets of the initial set of vertices, etc.

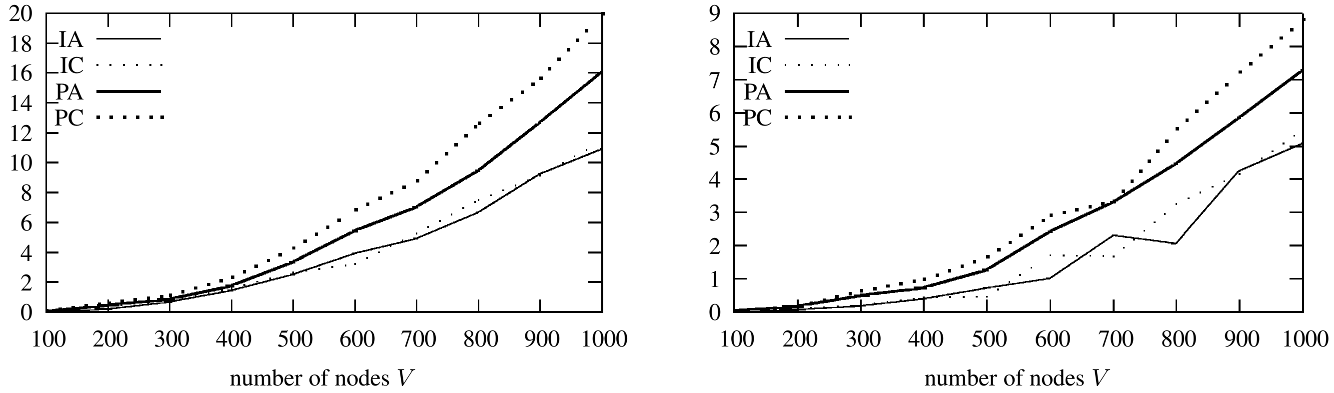

Figure 2. Benchmarking creation (left) and deletion (right)

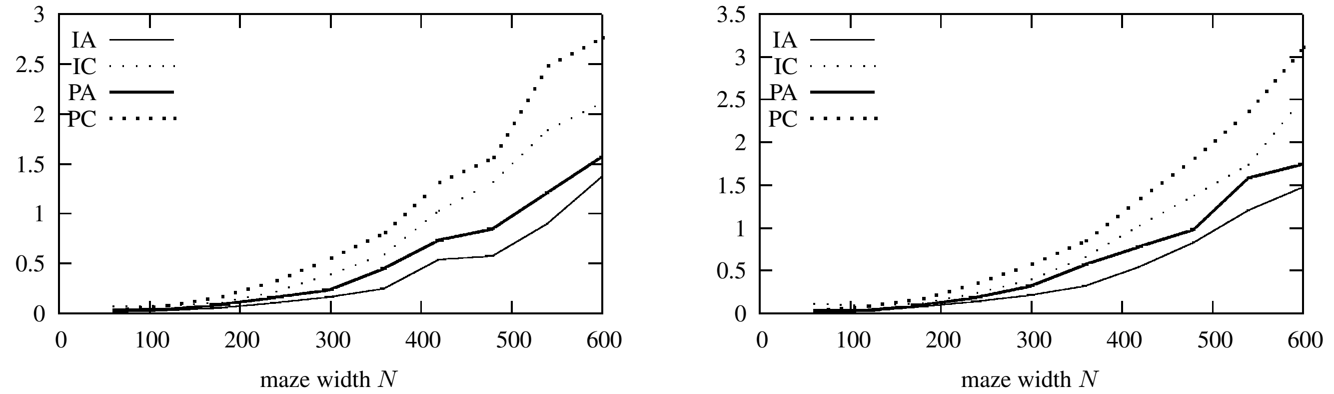

Figure 3. Benchmarking DFS (left) and BFS (right)

	language	persistent/imperative	generic algorithms	signature*s*	graph data structures
LEDA	C++	I	⊘	⊘	1
GTL	C++	I	⊘	⊘	1
MLRisc	SML	I	⊘	⊘	1
FGL	Haskell	P	⊘	⊘	1
Baire	OCAML	P/I	—	⊘	8
JDSL	Java	I	√	√	1
BGL	C++	I	√	√	1
OCAMLGRAPH	OCAML	P/I	√	√	19

Figure 4. Comparison with other graph libraries

A mechanism of behavioral views, similar to the principles of aspect programming, allows to connect actions to the modifiers of a graph. It is therefore possible to perform some given action before the addition of a new vertex, for instance. The library also contains many algorithms implemented as functors to be abstracted over specific operations independent of the graph data structure. For instance, the flow algorithm of MLRISC is parameterized by a structure of Abelian group, as OCAMLGRAPH's ones presented in Section 4.2.

LEDA [14] This C++ library provides several data structures (graphs, lists, sets, dictionaries, trees, partitions, priority queues, etc.) and several algorithms related to graphs, geometry, networks and graphics. LEDA's graphs are imperative. They may be directed or not, planar or not, and vertices and edges may be parameterized by user types. Macros are provided to iterate over vertices and edges. Many operations are implemented, such as access functions, modifiers, graph generators and property tests (cyclicity, planarity, convexity, etc.). Unfortunately, the algorithms are not generic and only one implementation is provided (using double-linked lists for vertices and edges).

FGL [3, 9] This HASKELL library — the SML version does not seem to be maintained anymore — provides an inductive representation of graphs and a set of graph algorithms for this data structure. FGL's graphs are multi-graphs, directed or not, persistent, with vertices and edges labeled with any type. The graph data structure is implemented using purely applicative arrays and AVLs. This library is the only one with OCAMLGRAPH and BAIRE (see below) to provide persistent graphs. However, the purely applicative philosophy of FGL makes a huge difference with our approach. Indeed, OCAMLGRAPH includes out-of-the-box imperative algorithms over graphs, thanks to a common interface between persistent and imperative graphs.

BGL [2] This C++ library provides generic algorithms implemented as *templates* (a purely syntactic counterpart of OCAML's functors). The parameters of these templates describe the minimal interface that the graph data structure must implement for the algorithm to apply. This library is the only one, with OCAMLGRAPH, to implement such generic algorithms. However, only two graph data structures are provided and the interface of BGL's graphs is less complete than OCAMLGRAPH's one. For instance, the labeling of vertices and edges is not part of the interface and is instead realized as external dictionaries passed to the algorithms as additional parameters.

BAIRE [1] This OCAML library provides 8 different implementations of graph data structures, without any code sharing (functors are not used in BAIRE). Each data structure implements directed graphs with labeled edges (there is no unlabeled graphs). Operations over vertices and edges are provided, and so are iterators and observers. BAIRE does not provide any graph algorithm.

GTL [5] This C++ library provides a unique imperative data structure for directed and unlabeled graphs (undirected graphs can be obtained by "canceling" the direction of edges). As in BGL the labeling is realized by external dictionaries. GTL provides several graph algorithms which only apply to its own data structure.

8. Conclusion

We have presented OCAMLGRAPH, a generic graph library for OCAML providing several graph data structures and, independently, several graph algorithms. This genericity is obtained using OCAML's module system and especially its functors which allow to share large pieces of code.

To our knowledge, there is no library for any applicative language as generic as OCAMLGRAPH. Regarding imperative languages, graph libraries are rarely as generic and, anyway, never provide as many different data structures.

Since its first release (Feb. 2004), the number of OCAMLGRAPH's users is steadily increasing 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] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

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

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

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