Le foncteur sonne toujours deux fois

S. Conchon[1] & J.-C. Filliâtre[1] & J. Signoles[1]

1: Laboratoire de Recherche en Informatique – CNRS 8623
Bât 490 Université Paris Sud 91405 ORSAY CEDEX, France
`{conchon,filliatr,signoles}@lri.fr`

Résumé

Cet article présente OCAMLGRAPH, une bibliothèque générique de graphes pour le langage de programmation OCAML. L'originalité de cette bibliothèque est de proposer d'une part un grand nombre de structures de données différentes pour représenter les graphes — graphes orientés ou non, structures persistantes ou modifiées en place, sommets et arcs avec ou sans étiquettes, marques sur les sommets, etc. — et d'autre part des algorithmes sur les graphes écrits indépendamment de la structure de données représentant les graphes. Le codage de ces deux aspects originaux a été rendu possible par une utilisation massive du système de modules d'OCAML et notamment de ses fonctions, les foncteurs.

1. Introduction

Trouver une bibliothèque de graphes pour son langage de programmation préféré n'est souvent pas très difficile. Mais pouvoir utiliser les algorithmes fournis par cette bibliothèque sur sa propre structure de graphe ou attendre de cette bibliothèque qu'elle permette de créer des graphes persistants, non-orientés dont les sommets et les arêtes sont étiquetés par autre chose que des entiers devient plus difficile.

Cet article présente OCAMLGRAPH[1], une bibliothèque générique de graphes pour le langage de programmation OCAML [7]. Au delà de la simple présentation de l'interface de cette bibliothèque, cet article est surtout l'occasion de montrer comment le langage OCAML a été utilisé pour introduire deux aspects originaux pour une telle bibliothèque. Le premier consiste à fournir non pas une structure de données de graphes mais de multiples structures de données énumérant toutes les variations possibles autour d'un certain nombre de critères — graphes orientés ou non, structures persistantes ou modifiées en place, sommets et arcs avec ou sans étiquettes, marques sur les sommets, etc. — mais regroupées sous des interfaces communes. La seconde originalité consiste à fournir un grand nombre d'algorithmes sur les graphes indépendamment de la structure de graphes sous-jacente. Ceux-ci peuvent être alors instanciés sur les nombreuses structures de données fournies par la bibliothèque mais également sur d'autres structures apportées par l'utilisateur, si tant est qu'elles satisfassent à des interfaces minimales.

Ces deux originalités sont rendues possibles par une utilisation massive du système de modules d'OCAML. Dans le premier cas, il s'agit d'éviter la duplication de code entre les différentes variantes de structures de données, une règle d'or en programmation et un impératif ici vu le nombre très élevé (17) de structures très semblables mais toutes différentes. Dans le second cas, il s'agit tout simplement de parvenir à écrire des algorithmes indépendamment des structures de données sous-jacentes, avec le plus de généricité possible mais avec un soucis constant d'efficacité du code final. Dans les deux

---

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

cas ce sont les fonctions du système de modules d'OCaml, appelées *foncteurs* dans le jargon de la programmation fonctionnelle, qui apportent la solution technique. Cet article a pour but d'expliquer ces deux utilisations très différentes des foncteurs — d'où le titre accrocheur — et il s'agit donc plus d'un article de génie logiciel que d'un article présentant une bibliothèque.

Le plan de cet article est le suivant. La section 2 présente brièvement le système de modules d'OCaml. La section 3 présente ensuite la conception de la signature commune à toutes les structures de données de graphes et montre comment le code de ses différentes réalisations a pu être factorisé. La section 4 décrit les algorithmes proposés dans Ocamlgraph et comment leur généricité vis-à-vis de la structure de graphes à pu être obtenue. Enfin la section 5 compare Ocamlgraph avec un certain nombre de bibliothèques existantes.

2. Le système de modules d'OCaml

Dans cette section, nous rappelons les caractéristiques du système de modules d'OCaml pour le lecteur qui n'en serait pas familier. Il s'agit d'un langage à part entière au dessus du langage de base d'OCaml, remplissant uniquement des fonctions de génie logiciel : compilation séparée, structuration de l'espace de noms, encapsulation, et généricité du code. Ce langage de modules pourrait être intégralement supprimé statiquement, et se trouve être en réalité indépendant du langage de base [11].

C'est un langage fonctionnel d'ordre supérieur fortement typé dont les termes sont appelés *modules*. Les briques de base sont les *structures*, unités regroupant entre eux des types, valeurs, exceptions et/ou définitions de modules. Par exemple, une implantation naïve sous forme de matrices d'adjacence des graphes orientés, étiquetés sur les arcs et modifiables en place peut être la suivante :

```
module Graph = struct
  type label = int
  type t = label array array
  let create n = Array.make_matrix n n 0
  let add_edge g v1 v2 l = g.(v1).(v2) <- l
  let iter g f v = Array.iter f g.(v)
end
```

Les types des structures, appelés *signatures*, permettent de restreindre la visibilité des composantes d'une structure en lui adjoignant une *interface*[2] et de masquer la définition de certains types (on parle alors de *type abstrait*). Ainsi, une signature possible masquant les types des graphes et des étiquettes de `Graph` est :

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

Les fonctions de ce langage, appelées *foncteurs*, permettent d'écrire des modules paramétrés par d'autres modules et de les appliquer ensuite à des modules particuliers. L'apport des foncteurs en terme de génie logiciel est important lorsqu'il s'agit de paramétrer un *ensemble* de types et de fonctions par un autre *ensemble* de types et de fonctions, de manière cohérente. Par exemple, pour rester indépendant de l'implantation de la structure de graphe, l'algorithme de Dijkstra, pour des graphes dont les arcs sont étiquetés par des entiers, peut être « fonctorisé » de la manière suivante :

---

[2] au sens de Modula et non de Java

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

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

L'annotation `with type` permet d'unifier ici le type abstrait `label` de la signature `S` avec le type `int`. D'autre part, la signature `S` exigée pour l'argument du foncteur ne contient que ce qui est nécessaire à l'implantation de l'algorithme. On pourra cependant appliquer ce foncteur à tout module dont la signature contient *au moins* `S`, c'est-à-dire qui est *sous-type* de `S`.

Enfin, il est possible de construire des aggrégats de signatures ou de modules à l'aide de la construction `include` qui peut être vue naïvement comme une inclusion textuelle.

3. Structures de données

Fournir de multiples structures de données de graphes est un des deux aspects originaux d'OCAMLGRAPH. En effet, un graphe peut être orienté ou non, ses sommets et ses arcs peuvent être avec ou sans étiquettes et la structure de données sous-jacente peut être *persistante* ou *modifiable en place* (*impérative)* : pas moins de seize structures de données différentes doivent donc être fournies pour rendre compte de ces diverses combinaisons. Par conséquent, il convient, d'une part, d'unifier au maximum les interfaces de ces structures de données pour permettre leur compréhension par l'utilisateur et, d'autre part, de factoriser le plus possible leurs implantations de manière à limiter la taille du code, et donc les bogues.

3.1. Les interfaces

Les interfaces des diverses structures de données de graphes sont regroupées dans le module `Sig` et sont essentiellement au nombre de trois : une interface `G` commune à toutes les structures de données, une interface `P` commune à toutes les structures persistantes (*i.e.* immuables d'un point de vue observationnel) et une interface `I` commune à toutes les structures impératives. L'interface commune `G` contient toutes les opérations d'accès à la structure du graphe. Les interfaces `P` et `I` étendent `G` en ajoutant les fonctions de création et de modification du graphe, nécessairement différentes selon le caractère persistant ou impératif.

L'interface commune `G` introduit un type abstrait `t` pour les graphes :

```
module type G = sig
  type t                                             (* le type des graphes *)
```

Le type des sommets est également un type abstrait[3]. Vu que l'on ne préjuge pas du caractère étiqueté ou non des sommets, on introduit également un type pour les étiquettes des sommets (on aurait pu choisir un type de sommets polymorphe mais le polymorphisme se marrie mal avec les foncteurs). Enfin, la plupart des algorithmes sur les graphes ont besoin de stocker des sommets dans des structures de données de type ensembles ou dictionnaires et il est pour cela très utile que le type des sommets

---

[3] Le type des sommets est abstrait *a priori* mais une réalisation particulière de la signature `G` peut tout à fait déclarer un type de sommets manifeste.

soit équipé de fonctions de comparaison et de hachage adéquates. On regroupe tout cela dans un sous-module `V` :

```
module V : sig
  type t                         (* le types des sommets *)
  type label                              (* le type des étiquettes des sommets *)
  val create : label -> t
  val label : t -> label
  val compare : t -> t -> int
  val hash : t -> int
  val equal : t -> t -> bool
end
```

L'intérêt d'introduire un sous-module est double : d'une part il évite de polluer l'espace de noms et d'autre part, il permet à ce sous-module d'être directement l'argument d'un foncteur tel que `Map.Make` ou `Hashtbl.Make`. On introduit tout de même un alias pour le type `V.t` pour plus de lisibilité par la suite :

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

Pour les graphes dont les sommets ne sont pas étiquetés, il suffira d'identifier les types `V.t` et `V.label` et de réaliser les deux fonctions `V.create` et `V.label` par l'identité.

De même on introduit un sous-module `E` pour les arcs. Là encore on suppose que les arcs peuvent être étiquetées et un arc est donc construit à partir de deux sommets et d'une étiquette. Enfin, comme pour les sommets, on équipe le type des arcs d'une fonction de comparaison.

```
module E : sig
  type t
  type label
  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
```

Pour des arcs non étiquetés, il suffira de prendre pour `E.t` le type `V.t * V.t` et d'ignorer le type `E.label` (en le réalisant par exemple par le type `unit`).

Viennent ensuite un grand nombre de fonctions de tests, de fonctions d'accès et d'itérateurs sur les sommets et les arcs dont la signification est immédiate :

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

La partie essentielle de l'interface réside dans les fonctions donnant accès à la relation d'adjacence dans le graphe. On trouve par exemple une fonction donnant les successeurs d'un sommet sous la forme d'une listes de sommets :

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

En pratique, cependant, il sera plus utile de pouvoir itérer une fonction sur les successeurs (ou les prédécesseurs), afin d'éviter de construire des liste pour immédiatement les déstructurer. Cet ensemble d'itérateurs complète la signature `G` :

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

Comme nous le montrerons dans la section 4, l'écriture d'un algorithme sur les graphes ne nécessite généralement qu'un très petit sous-ensemble de ces opérations. Grâce au sous-typage des signatures, une réalisation de l'interface `G` sera également une interface adéquate pour nos algorithmes.

L'interface `P` des graphes persistants étend alors l'interface `G` par un ensemble de fonctions de création de graphes :

```
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
```

De même l'interface `I` étend l'interface `G` par un ensemble de fonctions de création et de modification de graphes impératifs :

```
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
```

On notera la présence d'une fonction `copy` qui n'a pas de correspondant dans l'interface `P`.

Dans la section 4.4 nous montrerons comment il est possible de donner une interface commune à la construction de graphes, qu'ils soient persistants ou impératifs, afin de factoriser du code construisant des graphes. Mais il aurait été malheureux de le faire ici, car le caractère persistant ou impératif doit apparaître clairement à l'utilisateur et pour cela transparaître dans la signature.

En réalité, le module `Sig` introduit une quatrième et dernière signature de graphes, `IA`, pour les structures impératives où les sommets sont munis de *marques* entières que l'on peut modifier en place :

```
module type IA = sig
  include I
  module Mark : sig
    val clear : t -> unit                        (* met toutes les marques à 0 *)
    val get : vertex -> int
```

```
    val set : vertex -> int -> unit
  end
end
```

Une telle signature est utile lorsque l'on veut proposer un codage efficace d'un algorithme basé sur un marquage particulier des sommets, *a fortiori* si ces marques peuvent être ensuite exploitées par l'utilisateur. L'exemple typique est celui d'un parcours en profondeur.

On notera enfin qu'à aucun moment il n'a été nécessaire d'expliciter le caractère orienté ou non des graphes. Seule l'*implantation* le définira.

3.2. Le partage du code

Les implantations des diverses structures de données sont partagées entre les modules `Persistent` et `Imperative`. Chacun de ces deux modules se divise en deux sous-modules appelés `Graph` (graphes non orientés) et `Digraph` (graphes orientés). Ces derniers regroupent chacun quatre foncteurs ; chacun d'eux correspondent à des graphes concrets ou abstraits[4] possédant ou non des étiquettes sur ses arcs (un sommet est toujours étiqueté). Ils sont paramétrés par un module `V` correspondant aux sommets du graphe et, dans le cas des graphes étiquetés aux arcs, par un module `E` correspondant aux arcs. Par exemple, le foncteur `Persistent.Digraph.Abstract` implante les graphes persistants non orientés, abstraits et sans étiquette. Sa signature est la suivante :

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

Dans la suite de cette section, nous montrons comment les foncteurs sont utilisés pour factoriser un maximum de code entre ces seize structures de données. Il est important de noter que le code présenté ici n'est ni visible ni utile pour l'utilisateur de la bibliothèque.

Le module `Per_imp`, interne à OCAMLGRAPH, regroupe un certain nombre de foncteurs réalisant les opérations sur les graphes pour lesquelles il est possible de s'abstraire du caractère persistant ou impératif. Chaque foncteur est dédié à une tâche particulière comme poser des étiquettes sur les arcs (foncteur `Labelled`), rendre le graphe abstrait (foncteur `Make_Abstract`) ou coder les opérations sur les prédécesseurs (foncteur `Pred`). Coder une structure de données de graphes particulière revient alors à assembler les différents foncteurs comme dans un jeu de construction et à coder les quelques opérations qui ne peuvent pas être factorisées.

S'abstraire du caractère persistant ou impératif Dans notre bibliothèque, les graphes sont représentés sous la forme d'une table associant à chaque sommet du graphe l'ensemble de ses successeurs (auxquels est adjoint l'étiquette de l'arc correspondant si nécessaire). Ces tables d'association sont persistantes ou impératives selon le type de graphes considérés. En OCAML, cela se traduit par des opérations de types différents. Ainsi, la fonction d'ajout dans une table de hachage ne retourne aucune valeur (car la modification est faite en place) alors que celle d'ajout dans une table persistante retourne une nouvelle table. Mais on peut toutefois donner une signature commune à ces tables persistantes et impératives en introduisant un type explicite `'a return` de la manière suivante :

```
module type HM = sig
  type 'a return                          (* type de retour de [add] et [remove] *)
  type 'a t
  type key
```

---

[4] Un graphe abstrait masque le type (et donc l'implantation) des sommets et des arcs à l'utilisateur et permet une implantation optimisée de certaines opérations.

```
  val add: key -> 'a -> 'a t -> 'a return
  val mem: key -> 'a t -> bool
  val find: key -> 'a t -> 'a
  (* ... *)
end
```

Il est alors possible d'obtenir une implantation persistante de `HM` basée sur le module `Map` d'OCAML en instanciant le type `'a return` par le type des tables persistantes :

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

Et il en est de même pour une version impérative basée sur le module `Hashtbl` d'OCAML en instanciant le type `'a return` par `unit` :

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

Dès lors, en utilisant une implantation de `HM` ainsi qu'une implantation des ensembles, il est possible d'écrire un foncteur implantant les opérations communes à tous les types de graphe :

```
module Minimal(S: Set.S)(HM: HM) = struct
  let mem_vertex g v = HM.mem v g
  let out_degree g v =
    S.cardinal (try HM.find v g with Not_found -> invalid_arg "out_degree")
  (* ... *)
end
```

Foncteurs comme briques d'assemblage Les opérations sur les prédécesseurs se programment toujours de la même manière. Elles sont regroupées dans le foncteur `Pred` :

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

En outre, dans le cas des graphes non orientés, les opérations sur les prédécesseurs peuvent être optimisées en tenant compte du fait que l'ensemble des prédécesseurs d'un sommet s est égal à l'ensemble des successeurs de s. On peut alors redéfinir `Pred` :

```
module Pred(S: (* ... *) ) = struct
  let fold_pred = S.fold_succ
  let pred = S.succ
  (* ... *)
end
```

Un certain nombre d'opérations, comme itérer sur les arcs d'un graphe, dépendent de la présence ou non d'étiquettes sur les arcs du graphe. Ceci entraîne l'implantation de deux foncteurs `Labelled` et `Unlabelled` (non détaillés dans cet article).

Il est possible de rendre un graphe abstrait à partir de sa version concrète et, ainsi, d'optimiser certaines opérations. C'est le rôle dédié au foncteur `Make_Abstract` :

```
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                                        (* optimisation *)
  (* ... *)
end
```

Enfin, il est possible d'implanter la version concrète du module `V` et la table d'association l'utilisant à l'aide d'un foncteur d'ordre supérieur :

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

Assemblage des briques A l'aide de l'ensemble de ces foncteurs, il est possible d'implanter de manière générique — et aisément ! — la majeure partie des modules `Digraph` et `Graph` :

```
module Make(F: functor(X: (* ... *) ) -> HM with type key = X.t) = struct
  module Digraph = struct
    module Concrete(V: (* ... *) ) = struct
      include ConcreteVertex(F)(V)                   (* les sommets sont concrets *)
      include Unlabeled(V)(HM)                  (* les arcs ne sont pas étiquetés *)
      include Minimal(S)(HM)                        (* opérations communes à tous *)
    end
    module ConcreteLabelled = (* ... *)
    module Abstract = (* ... *)
    module AbstractLabelled = (* ... *)
    end
  end
  module Graph = struct (* ... *) end
end
```

Les implantations persistantes et impératives sont obtenues en appliquant ce foncteur :

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

Pour obtenir les foncteurs visibles par l'utilisateur, il reste à étendre les foncteurs obtenus en ajoutant les quelques fonctions qui ne peuvent pas être factorisées :

```
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
```

4. Algorithmes

Nous présentons dans cette section la deuxième utilisation des foncteurs dans OCAMLGRAPH : la programmation générique d'algorithmes sur les graphes.

4.1. Programmation générique

Comme nous l'avons vu en partie 3, notre bibliothèque fournit de nombreuses implantations de graphe. Le problème se pose alors d'écrire des algorithmes pour toutes ces structures sans avoir à les réécrire pour chacune d'elles. Tout comme les foncteurs nous ont permis d'implanter ces structures de données avec un minimum de duplication de code, ils vont également nous permettre de programmer des algorithmes génériques à moindre frais.

L'idée de base est d'écrire un algorithme sans se soucier de l'implantation des graphes mais en se concentrant uniquement sur les opérations nécessaires et suffisantes à la manipulation de ces structures de données. L'algorithme devient alors naturellement un foncteur paramétré par ces opérations et la signature qu'elles engendrent est le type de son paramètre. Ces opérations correspondent en général à un sous-ensemble de celles fournies par OCAMLGRAPH mais, comme nous le verrons dans les exemples donnés par la suite, certains algorithmes ont besoin d'opérations spécifiques, indépendantes de la structure du graphe, qui seront alors fournies comme des paramètres supplémentaires au foncteur.

Cette « fonctorisation » des algorithmes permet, d'une part, d'ajouter facilement des algorithmes à la bibliothèque et, d'autre part, d'utiliser facilement les algorithmes d'OCAMLGRAPH sur des structures de données non fournies par la bibliothèque.

4.2. Exemple : algorithmes de flots

Nous illustrons l'aspect générique des algorithmes d'OCAMLGRAPH à travers deux exemples pour calculer le flot maximal d'un réseau[5], l'algorithme de Ford et Fulkerson [9] et celui de Goldberg [10]. Ils se présentent sous forme de deux foncteurs dont les signatures sont les suivantes :

```
module Ford_Fulkerson (G: G_FORD_FULKERSON)(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
```

---

[5] *i.e.* un graphe orienté possédant deux sommets distingués appelés *source* et *puits*

```
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
```

Les corps de ces foncteurs implantent une fonction `maxflow` de même signature prenant en paramètre un réseau, une source et un puits et retournant une paire `(f,delta)` où `f` associe à chaque arc du réseau son nouveau flot et où `delta` représente la différence entre le flot maximal calculé et le flot initial du réseau.

La signature du premier paramètre `G` de ces deux foncteurs décrit les opérations minimales du graphe pour l'implantation de `maxflow`. Puisque les deux algorithmes implantés par ces foncteurs sont différents, les signatures `G_FORD_FULKERSON` et `G_GOLDBERG` sont également différentes. Par exemple l'algorithme de Goldberg nécessite d'itérer sur tous les sommets et arcs du graphe tandis que celui de Ford et Fulkerson a besoin d'itérer sur les arcs issus d'un sommet, comme le montrent leurs signatures respectives :

```
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
```

La signature du second paramètre `F` est commune aux deux foncteurs. Elle rassemble les opérations pour manipuler les étiquettes des arcs comme des flots :

```
module type FLOW = sig
  type label                              (* type des étiquettes d'un arc *)
  type t                             (* type des flots passant sur un arc *)
  val flow : label -> t                                    (* flot actuel *)
  val max_capacity : label -> t                     (* capacité maximale *)
  val min_capacity : label -> t                     (* capacité minimale *)
  val add : t -> t -> t                                       (* addition *)
  val sub : t -> t -> t                                   (* soustraction *)
  val zero : t                                          (* élément neutre *)
  val compare : t -> t -> int                   (* fonction de comparaison *)
end
```

Cette signature `FLOW` s'implante facilement quelque soit le type de données. Par exemple, une implantation possible pour un graphe dont les étiquettes des arcs seraient des entiers correspondant à la capacité maximale de chaque arc et dont le flot initial serait nul est la suivante :

```
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. Autre exemple : parcours de graphes

Nous donnons ici un second exemple de programmation générique avec les algorithmes classiques de parcours en profondeur et en largeur. Pour parcourir un graphe, il y a besoin de très peu d'opérations. Principalement, il faut pouvoir itérer une fonction sur tous les successeurs d'un sommet. Il faut également pouvoir parcourir l'ensemble des sommets pour parcourir toutes les composantes connexes. Enfin il faut pouvoir marquer les sommets visités. Concernant ce dernier point, et en l'absence de toute hypothèse sur les sommets du graphe, le plus simple est d'utiliser une table de hachage contenant les sommets déjà visités. Pour cela, il suffit que le type des sommets soit équipé de fonctions `hash` et `equal` convenables. L'interface minimale requise pour les algorithmes de parcours de graphes est donc la suivante :

```
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
```

Dès lors les algorithmes de parcours se présentent sous la forme de foncteurs ressemblant à ceci :

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

En réalité, plusieurs variantes de chaque parcours sont offertes : composante à partir d'un sommet donné uniquement, application de la fonction après les successeurs plutôt qu'avant, etc. Le foncteur de parcours en profondeur propose également une fonction de détection de cycle. Enfin, il y a un foncteur spécialisé pour les graphes équipés de marques (voir la signature `IA` au paragraphe 3.1), car celles-ci peuvent être alors utilisées de manière directe pour noter les sommets déjà visités.

Les fonctions de parcours de graphes sont également proposées dans des versions « pas à pas », à l'instar des itérateurs que l'on trouve plutôt dans des contextes de programmation impérative (telles que les classes réalisant l'interface `Enumeration` en JAVA). Un tel itérateur se présente sous la forme suivante :

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

Le type abstrait `iterator` représente un état de l'itérateur, ou si l'on préfère un point dans l'itération en cours. On obtient le point de départ avec la fonction `start`. Dès lors, on peut avancer dans l'itération avec la fonction `step` ou récupérer le sommet en cours de visite avec `get`. Ces deux fonctions lève l'exception `Exit` lorsque l'on se trouve dans l'état final. Cette séparation en deux fonctions distinctes `step` et `get` constitue une différence importante avec les itérateurs « pas à pas » que l'on trouve habituellement en programmation impérative, où une seule fonction retourne l'élément courant *et* avance l'itérateur. D'autre part, et c'est là une différence encore plus importante, le type `iterator` est un type *persistant* : l'itérateur n'est pas modifié par la fonction `step`. Dès lors, il est très facile d'utiliser ces itérateurs dans des algorithmes de *backtracking* impliquant des graphes.

Voici par exemple comment peut être codée la k-coloration d'un graphe `g` sur la base d'un parcours en largeur pas à pas `Bfs`. Le graphe possède des marques que l'on utilise pour indiquer la couleur (0 signifiant qu'il n'y a pas encore de couleur assignée).

```
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 -> ()
```

Le gain de l'itérateur pas à pas persistant est double : non seulement on a pu utiliser une fonction de parcours existante sans avoir besoin de la réécrire au sein même de l'algorithme de coloration, mais la persistance de l'itérateur a également permis d'éviter une écriture explicite du *backtrack* dans le code ci-dessus.

4.4. Constructions de graphes

Nous avons montré au paragraphe 3.1 que les opérations de constructions de graphes différaient entre les graphes persistants et les graphes impératifs, ce qui est parfaitement justifié. Cependant, de même que nous avons écrit des algorithmes sur les graphes de manière générique, on peut souhaiter *construire* des graphes de manière générique, *i.e.* indépendamment de la structure de données les représentant. On peut ainsi vouloir définir des opérations telles que l'union de deux graphes, la clôture transitive d'un graphe ou encore construire des graphes classiques (graphe complet à n sommets, graphe de de Bruijn d'ordre n, graphe des diviseurs, etc.) ou des graphes aléatoires, etc. Dans tous ces cas, le caractère persistant ou impératif des graphes n'est pas significatif mais la différence d'interfaces s'oppose à la généricité.

Pour remédier à cela, on introduit un module `Builder`. Il définit une interface de construction commune aux graphes persistants et impératifs :

```
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
```

Il est immédiat d'implanter une telle signature à partir d'une structure persistante ou d'une structure impérative :

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

Il est important de noter que, pour une structure impérative, les graphes retournés par les fonctions `add_vertex`, `add_edge` et `add_edge_e` ne sont pas significatifs. Il y a même là une illusion trompeuse de persistance. Mais cette interface commune se limite à ce module `Builder` et comporte un avertissement adéquat.

Dès lors, il est facile de coder un algorithme générique de construction de graphes. On commence par écrire une version très générique prenant en argument un module de type `Builder.S` :

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

puis on peut trivialement proposer deux variantes de ce foncteur pour les graphes persistants et les graphes impératifs avec les deux lignes suivantes :

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

Ainsi l'utilisation du module `Builder` est entièrement cachée à l'utilisateur de la bibliothèque.

5. OCAMLGRAPH et les autres bibliothèques de graphes

Nous présentons dans cette section les fonctionnalités de quelques bibliothèques de graphes afin de donner au lecteur des éléments de comparaison sur les possibilités qu'offrent ces bibliothèques et ce que nous fournissons avec OCAMLGRAPH.

Loin d'être exhaustive, notre liste présente des bibliothèques écrites dans (ou pour) des langages de programmation permettant une certaine forme de programmation générique. Par conséquent, les critères de comparaison que nous avons retenus portent principalement sur la variété de structures de données de graphes et d'opérations associées (itérateurs, modificateurs etc.) proposée dans ces bibliothèques et sur la généricité des algorithmes fournis.

JDSL [4] Cette bibliothèque JAVA fournit des implantations et des algorithmes pour plusieurs structures de données (listes, vecteurs, files à priorité, dictionnaires et graphes). En ce qui concerne les graphes, les interfaces définies dans JDSL regroupent — dans des signatures distinctes — les opérations d'accès en lecture et en écriture à la structure de données sous-jacente : On trouve des *accesseurs* pour l'accès en temps constant aux éléments du graphe, des *itérateurs* pour parcourir la structure de données et des *modificateurs* pour ajouter ou supprimer des sommets et des arcs. L'utilisation du système d'interfaces et des classes abstraites de `Java` rend les algorithmes de JDSL très génériques. La bibliothèque ne propose cependant qu'une seule implantation de graphe par listes d'adjacences et on ne trouve pas de notion de graphe persistant dans les signatures des interfaces. De plus, aucun type de données ne permet de distinguer les graphes orientés des graphes non-orientés (arêtes et arcs pouvant coexister dans un même graphe). Enfin, les sommets et les arcs peuvent être étiquetés par des informations de type `Object`, mais cette technique de programmation, bien connue des programmeurs JAVA, offre une forme de polymorphisme non sûre qui oblige à effectuer des vérifications de typage au moment de l'exécution.

MLRISC [6] Ce Back-End de compilateur paramétrable fournit une bibliothèque de graphes pour SML ne contenant qu'une unique signature de multi-graphes orientés ainsi qu'une unique implantation impérative. Les sommets et les arcs d'un graphe peuvent être étiquetés par des informations de type quelconque. Des modificateurs et des itérateurs permettent respectivement de mettre à jour et de parcourir la structure de données sous-jacente. On trouve également des combinateurs pour construire des *vues* différentes d'un même graphe. On peut par exemple construire une vue d'un graphe où les arcs sont inversés, où les sommets (resp. les arcs) sont renommés ou forment un sous-ensemble de l'ensemble des sommets (resp. des arcs) du graphe etc. Un système de vues *comportementales*, semblable aux principes de la programmation par aspect, permet d'attacher des actions aux modificateurs d'un graphe afin, par exemple, de réaliser telles ou telles actions avant l'ajout d'un sommet. La bibliothèque contient également de nombreux algorithmes dont certains sont implantés sous forme de foncteurs afin de s'abstraire d'opérations spécifiques, indépendantes de la structure de graphes (par exemple, l'algorithme de flot de MLRISC est, comme ceux d'OCAMLGRAPH présentés en section 4.2, paramétré par une structure de groupe abélien).

LEDA [12] Cette bibliothèque en C++ fournit de nombreuses structures de données (graphes, listes, ensembles, dictionnaires, arbres, partitions, file de priorités, lignes, points, surfaces etc.) ainsi que de nombreux algorithmes sur les graphes, la géométrie, les réseaux et le graphisme. Les graphes de LEDA sont impératifs. Ils peuvent être orientés ou non, paramétrés par un type de sommet et un type d'arc quelconques, statiques ou planaires. Des macros permettent d'itérer sur les sommets et sur les arcs. On trouve également de nombreuses fonctions d'accès, des modificateurs, des générateurs de graphes et des tests de toutes sortes (cyclicité, planarité, convexité, etc.). Malheureusement, les algorithmes de cette bibliothèque ne sont pas génériques et une seule implantation est fournie (à l'aide de deux listes doublement chaînées de sommets et d'arcs).

FGL [3, 8] Cette bibliothèque de graphes pour HASKELL (la version pour SML ne semble plus être maintenue) propose une représentation inductive des graphes et fournit un ensemble d'algorithmes fonctionnels pour cette nouvelle structure de données. Les graphes de FGL sont des multi-graphes, orientés ou non, persistants, dont les sommets et les arcs sont étiquetés avec des informations de type quelconque. L'implantation des graphes est réalisée par des tableaux fonctionnels ou des arbres AVL. Cette bibliothèque est la seule, avec OCAMLGRAPH et BAIRE (voir ci-dessous), à proposer des graphes persistants. Cependant, la «philosophie» purement fonctionnelle de FGL est bien différente de celle avec laquelle nous avons conçu notre bibliothèque. Par exemple, l'interface de nos graphes persistants partage une importante signature avec nos graphes impératifs et les algorithmes que l'on fournit sont directement inspirés des algorithmes *impératifs* de la littérature.

BGL [2] Cette bibliothèque de graphes pour C++ propose des algorithmes génériques implantés à l'aide de *templates* (forme purement «syntaxique» des foncteurs d'OCAML). Les paramètres de ces templates décrivent l'interface minimale que la structure de données de graphes doit remplir pour appliquer l'algorithme. Cette bibliothèque est la seule, avec OCAMLGRAPH, à proposer de tels algorithmes génériques. Cependant, on ne trouve que deux implantations de graphes et la liste des interfaces définies dans BGL est moins complète que celle de notre bibliothèque : on ne retrouve pas, par exemple, d'interfaces permettant aux algorithmes de profiter du caractère persistant d'une structure de données. De plus, l'étiquetage des sommets ou des arcs est réalisé à l'aide de dictionnaires passés comme paramètres aux algorithmes qui en ont besoin, et ne fait donc pas parti de la signature des graphes.

BAIRE [1] Cette bibliothèque de structures de données pour CAML fournit huit implantations différentes de graphes sans aucun partage de code (la bibliothèque n'utilisant pas le système de foncteurs d'OCAML). Chaque structure implante des graphes orientés et étiquetés sur les arcs (il n'y a pas de graphes non étiquetés) et possède des opérations sur les arcs et les sommets ainsi qu'un ensemble d'itérateurs et d'observateurs. BAIRE ne fournit aucun algorithme sur les graphes.

GTL [5] Cette bibliothèque en C++ propose une unique structure de graphes impératifs, orientés (un graphe non-orienté est obtenu en «annulant» la direction des arcs d'un graphe orienté) et non-étiquetés. Tout comme BGL, l'étiquetage des sommets ou des arcs est réalisé à l'aide de dictionnaires. Enfin, la bibliothèque fournit plusieurs algorithmes pouvant être appliqués uniquement sur sa structure de données.

6. Conclusion

Nous avons présenté dans cet article OCAMLGRAPH, une bibliothèque de graphes pour OCAML, proposant de nombreuses structures de données de graphes ainsi que des algorithmes implantés indépendamment de ces structures. Cette généricité est réalisée grâce au système de modules d'OCAML, et tout particulièrement à ses foncteurs qui permettent une large factorisation du code au niveau des implantations tant des différentes structures de données que des algorithmes.

À notre connaissance, aucune bibliothèque d'un langage applicatif n'est aussi générique qu'OCAMLGRAPH. Celles des langages impératifs ne le sont que rarement et, quoi qu'il en soit, ne fournissent jamais une telle diversité de structures de données.

Il reste à comparer les performances d'OCAMLGRAPH à celles des bibliothèques existantes. Cependant l'attention portée à l'efficacité des structures de données utilisées dans OCAMLGRAPH autorise un certain optimisme, qui semble confirmé par les toutes premières expériences.

Références

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

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

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

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

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