Programmation Avancée

Modules

Armaël Guéneau

Programmation modulaire

Initialement un problème de génie logiciel : découper le code en unités

- **Simplification** : pouvoir développer un module avec une connaissance limitée des autres modules
- **Incrémentalité** : pouvoir modifier/remplacer un module sans modifier le système complet
- **Robustesse** : pouvoir "passer à l'échelle" ; correction du code dans la durée
- **Réutilisation** : composants réutilisables pour éviter la duplication de code
- **Test** : pouvoir tester séparément les modules

Programmation modulaire : concepts

- **Composant** : partie de programme prévue pour être indépendante $\rightarrow$ moyen de structuration
- **Implémentation** : code du composant
- **Interface** : description des parties du composant visibles par d'autres $\rightarrow$ moyen d'abstraction
- **Spécification** : comportement observable du composant, décrit en terme de l'interface

Programmation modulaire : concepts

- **Composant** : partie de programme prévue pour être indépendante $\rightarrow$ moyen de structuration
- **Implémentation** : code du composant
- **Interface** : description des parties du composant visibles par d'autres $\rightarrow$ moyen d'abstraction
- **Spécification** : comportement observable du composant, décrit en terme de l'interface

Barbara Liskov : "data abstraction" & "substitution principle"

$\rightarrow$ un changement d'implémentation devrait être **transparent** pour un utilisateur de l'interface d'un composant.

Exemple : tri modulaire

Composant 1 : **file de priorité** (interface)

```
type pqueue
val empty  : pqueue
val insert : int * pqueue -> pqueue
val deletemax : pqueue -> int * pqueue
```

Composant 2 : **procédure de tri** (implémentation)

```
let sort (a : int array) =
  let n = Array.length a in
  let s = ref empty in
  for i = 0 to n-1 do s := insert (a.(i), !s) done;
  for i = n-1 downto 0 do
    let x, s' = deletemax !s in
    a.(i) <- x; s := s'
  done
```

Programmation modulaire

Idées qui existent sous différentes formes dans différents langages : OCaml, SML, mais aussi Ada, Java, Scala, etc...

En OCaml : *système de modules*. Approche qui remonte à Modula puis développée avec Standard ML ("SML")

Mécanismes fournis comme un langage de modules, intégré au langage.

Les garanties d'abstraction / séparation entre composants sont fournies par le compilateur

Modules ML (OCaml/SML)

Jouent plusieurs rôles :

- gestion des noms
- abstraction
- réutilisation de code

Un sous langage essentiellement indépendant du langage de base

Références pour OCaml : le manuel (`https://ocaml.org/manual/`),
Chapitres 2, 9.10, 9.11, 10.{6,7,8,9,15,22}.

Fichiers et modules

chaque fichier est un module

si `arith.ml` contient :

```
let pi = 3.1415926
let round x = floor (x +. 0.5)
```

dans un autre fichier `main.ml` :

```
let x = float_of_string (read_line ())
let () =
  print_float (Arith.round (x /. Arith.pi));
  print_newline ()
```

Interfaces

on peut restreindre les valeurs exportées avec une **interface**

dans un fichier `arith.mli` :

```
val round : float -> float
```

`main.ml` ne compile alors plus :

```
File "main.ml", line 3, characters 33-41:
3 |   print_float (Arith.round (x /. Arith.pi));
                                     ^^^^^^^^
Error: Unbound value Arith.pi
```

Le langage de modules est également utilisable dans un même fichier

on peut imbriquer les modules à volonté

```
module Arith = struct
  let pi = 3.1415926
  let round x = floor (x +. 0.5)
end

let x = float_of_string (read_line ())
let () =
  print_float (Arith.round (x /. Arith.pi));
  print_newline ()
```

Langage de modules : exemple

```
module Frac = struct
  type t = int * int

  let reduce (a, b) =
    ...

  let create a b =
    assert (b <> 0); reduce (a, b)

  let add (a, b) (a', b') =
    reduce (a * b' + a' * b, b * b')
end
```

Accès aux éléments d'un module

- La notation "point" : `Frac.create`, `Frac.add`

- S'applique aussi aux champs des types enregistrements :

```
module Pos = struct
  type t = { x : int; y : int }
end
let p = { Pos.x = 3; Pos.y = 4 }
let p = { Pos.x = 3; y = 4 }
```

- Et aux constructeurs de types algébriques :

```
module Result = struct
  type ('a, 'b) t = Ok of 'a | Error of 'b
end
let ok = Result.Ok 42
let err = Result.Error "failure"
```

Ouverture d'un module

On peut "ouvrir" un module pour accéder aux déclarations de la structure associée :

```
open Frac
let f = add (create 1 2) (create 1 3)
```

Aussi possible localement avec `M.(...)` ou `let open M in ...` :

```
let f = Frac.(add (create 1 2) (create 1 3))
let f =
  let open Frac in
  add (create 1 2) (create 1 3)
```

Langage de modules : signatures

```
module type FRAC = sig
  type t = int * int
  val create : int -> int -> t
  val add : t -> t -> t
end

module Frac = struct
  type t = int * int
  let reduce (a, b) = ...
  let create a b =
    assert (b <> 0); reduce (a, b)
  let add (a, b) (a', b') = ...
end
```

Langage de modules : signatures

```
module type FRAC = sig
  type t = int * int
  val create : int -> int -> t
  val add : t -> t -> t
end

module Frac : FRAC = struct
  type t = int * int
  let reduce (a, b) = ...
  let create a b =
    assert (b <> 0); reduce (a, b)
  let add (a, b) (a', b') = ...
end
```

Le compilateur **vérifie statiquement** que l'implémentation satisfait l'interface

Langage de modules : signatures

One peut aussi directement utiliser une signature anonyme :

```
module Frac : sig
  type t = int * int
  val create : int -> int -> t
  val add : t -> t -> t
end = struct
  type t = int * int
  let reduce (a, b) = ...
  let create a b =
    assert (b <> 0); reduce (a, b)
  let add (a, b) (a', b') = ...
end
```

Langage de modules : pour récapituler

- "`module M = ...`" : déclaration de nom de module
  $\rightarrow$ analogue à "`let x = ...`" pour les valeurs
- "`struct ... end`" : « expression » de module anonyme
- "`module type S = ...`" : déclaration de nom de signature/interface
  $\rightarrow$ analogue à "`type t = ...`"
- "`sig ... end`" : signature anonyme

Remarque

L'analogie module/signature $\leftrightarrow$ valeur/type a ses limites

Un module contient à la fois une composante statique (déclarations de types) et dynamique (code)

À l'exécution la composante statique est effacée

Signatures : typage structurel

le typage des modules et signatures est **structurel** (contrairement à la plupart des types pour les valeurs)

ce qui importe c'est le contenu d'une signature, pas son nom

```
module M = struct
  let x = 3
  let y = "hello"
end

module type S1 = sig val x : int end
module type S2 = sig val y : string end

module M1 = (M : S1)  (* M1.x ok, M1.y non *)
module M2 = (M : S2)  (* M2.x non, M2.y ok *)
```

jusque là : gestion de noms

Hiérarchiser les noms de manière logique

On peut imbriquer les modules autant que l'on veut

Permet de réutiliser le même nom dans différents modules (p.ex. `map`)

Cacher les déclarations privées qui correspondent à des détails d'implémentation

Remarque

On peut déjà **cacher** des déclarations privées, sans utiliser le langage de modules !

Comment réécrire ce programme pour cacher `reduce` au reste du programme ?

```
let reduce (a, b) = ...

let create a b =
  assert (b <> 0);
  reduce (a, b)
```

Remarque

On peut déjà **cacher** des déclarations privées, sans utiliser le langage de modules !

Comment réécrire ce programme pour cacher `reduce` au reste du programme ?

```
let reduce (a, b) = ...

let create a b =
  assert (b <> 0);
  reduce (a, b)
```

```
let create a b =
  let reduce (a, b) =
    ...
  in
  assert (b <> 0);
  reduce (a, b)
```

Remarque (2)

```
let reduce (a, b) = ...

let create a b =
  assert (b <> 0);
  reduce (a, b)

let add (a, b) (a', b') =
  reduce (a * b' + a' * b,
          b * b')
```

Remarque (2)

```
let reduce (a, b) = ...

let create a b =
  assert (b <> 0);
  reduce (a, b)

let add (a, b) (a', b') =
  reduce (a * b' + a' * b,
          b * b')
```

```
let create, add =
  let reduce (a, b) = ... in

  let create a b =
    assert (b <> 0);
    reduce (a, b)
  in

  let add (a, b) (a', b') =
    reduce (a * b' + a' * b,
            b * b')
  in
  create, add
```

Remarque (2)

```
let reduce (a, b) = ...

let create a b =
  assert (b <> 0);
  reduce (a, b)

let add (a, b) (a', b') =
  reduce (a * b' + a' * b,
          b * b')
```

```
let create, add =
  let reduce (a, b) = ... in

  let create a b =
    assert (b <> 0);
    reduce (a, b)
  in

  let add (a, b) (a', b') =
    reduce (a * b' + a' * b,
            b * b')
  in
  create, add
```

Les signatures permettent d'avoir des fonctions privées au "toplevel"

abstraction

Exemple : module `Frac`

```
module Frac = struct
  type t = int * int

  let reduce (a, b) = ... (* plante si b=0 *)

  let create a b =
    if b = 0 then None
    else Some (reduce (a, b))

  let add (a, b) (a', b') =
    reduce (a * b' + a' * b, b * b')
end
```

Exemple : module `Frac`

Propriétés attendues par un client du module :

- On ne peut pas créer de fraction avec dénominateur égal à zéro
- Le module ne plante pas / ne lance pas d'exception

Invariants internes de l'implémentation du module :

- Une fraction a un dénominateur non nul
- Toutes les fractions manipulées sont réduites

Une première signature

```
module type FRAC1 = sig
  type t = int * int
  val create : int -> int -> t option
  val add : t -> t -> t
end

module Frac : FRAC1 = struct ... end
```

Une première signature

```
module type FRAC1 = sig
  type t = int * int
  val create : int -> int -> t option
  val add : t -> t -> t
end

module Frac : FRAC1 = struct ... end
```

Problème : un client peut créer à la main des valeurs de type `Frac.t`, *qui ne respectent pas les invariants internes*

- `(1, 0)`
- `(9, 6)`

Types abstraits

Solution : cacher la définition du type `t`. Il devient un type **abstrait**.

Un client sait que le type existe, mais ne connait pas sa définition.

```
module type FRAC2 = sig
  type t
  val create : int -> int -> t option
  val add : t -> t -> t
end

module Frac : FRAC2 = struct ... end
```

Types abstraits et types existentiels

Autrement dit, les types abstraits correspondent à une forme de types existentiels. `FRAC2` peut être vue comme le type :

$$\exists\tau.\ (\text{int} \to \text{int} \to \tau\ \text{option}) \times (\tau \to \tau \to \tau)$$

C'est une connexion qui peut être explicitée formellement, voir :

- *Abstract Types Have Existential Type*, Mitchell et Plotkin
- *F-ing modules*, Rossberg, Russo et Dreyer

La traduction étendue à l'ensemble du langage de modules devient assez complexe...

Types abstraits

Les types abstraits : un mécanisme **central** de la programmation en ML

abstraction : découpler représentation des données et modèle logique

$\rightarrow$ éviter de confondre des valeurs conceptuellement distinctes parce que leur représentation coïncide

faire du système de types son allié

$\rightarrow$ l'implémenteur du module peut associer des invariants de son choix aux valeurs du type abstrait

Le mécanisme qui permet de « passer à l'échelle »

Conséquence

deux implémentations équivalentes d'une même signature sont *indistinguables* par un client

```
module Frac_alt : FRAC2 = struct
  type t = Int of int | Frac of int * int
  let create = ...
  let add = ...
end
```

Les modules `Frac` et `Frac_alt` sont équivalents.

Question : peut on mélanger des `Frac.t` et des `Frac_alt.t` ?

p.ex : `let f = Frac_alt.add (Frac.create 1 2) (Frac.create 1 3)`

Exemples de modules avec types abstraits

Dans la bibliothèque standard, par exemple :

- `Queue`
- `Stack`
- `Buffer`
- `Hashtbl`...

On peut aller lire le code, mais pas s'appuyer sur les détails d'implémentation des modules

spécialisation de signatures

signatures : réutilisation et spécialisation

on peut vouloir réutiliser une signature en **spécialisant** certains types

```
module type Group = sig
  type t
  val unit : t
  val op : t -> t -> t
  val inv : t -> t
end
```

signatures : réutilisation et spécialisation

on peut vouloir réutiliser une signature en **spécialisant** certains types

```
module type Group = sig
  type t
  val unit : t
  val op : t -> t -> t
  val inv : t -> t
end

module ZGroup : Group = struct
  type t = int
  let op = (+) (* ... *)
end
```

signatures : réutilisation et spécialisation

on peut vouloir réutiliser une signature en **spécialisant** certains types

```
module type Group = sig
  type t
  val unit : t
  val op : t -> t -> t
  val inv : t -> t
end

module ZGroup : Group = struct
  type t = int
  let op = (+) (* ... *)
end

let x = ZGroup.op 1 2
(* Error:              ^  this expression has type int
          but an expression was expected of type ZGroup.t *)
```

signatures : réutilisation et spécialisation (2)

Solution 1 : définir `ZGroup` sans lui assigner la signature `Group`

signatures : réutilisation et spécialisation (2)

Solution 1 : définir `ZGroup` sans lui assigner la signature `Group`

Solution 2 : "`S with type t = foo`" spécialise `S` avec `t = foo`

```
module ZGroup : (Group with type t = int) = struct
  type t = int
  ...
end

let x : int = ZGroup.op 1 2
```

foncteurs

terminologie

ATTENTION : il existe un *autre* concept en programmation fonctionnelle appelé “foncteur”

ici on parle de “foncteurs OCaml” ou “foncteurs ML”, spécifiques aux systèmes de modules

foncteurs

"foncteur" : **module paramétré** par un ou plusieurs autres modules

ou encore : "fonction" au niveau du langage de modules

cas d'utilisation typique : structures de données génériques

- table de hachage paramétrée par les fonctions de hachage et égalité sur les clefs (`Hashtbl.Make`)
- structure d'ensemble paramétrée par la fonction de comparaison sur les éléments (`Set.Make`)

utilité des foncteurs ? un exemple

Implémentons des ensembles génériques, sans foncteur.

Essai n°1

interface :

```
type 'a t
val empty : 'a t
(* ajout d'un élément à l'ensemble *)
val add : 'a -> 'a t -> 'a t
(* test d'appartenance à l'ensemble *)
val mem : 'a -> 'a t -> bool
```

utilité des foncteurs ? un exemple

implémentation simple (peu efficace) : listes triées d'entiers

```
type 'a t = 'a list
let empty = []

let add x l =
  (* ... ? ... *)

let mem x l =
  (* ... ? ... *)
```

utilité des foncteurs ? un exemple

implémentation simple (peu efficace) : listes triées d'entiers

```
type 'a t = 'a list
let empty = []

let rec add x l =
  match l with
  | [] -> [x]
  | y :: l' -> if x <= y then x :: l
               else y :: add x l

let rec mem x l =
  match l with
  | [] -> false
  | y :: l' -> y <= x && (x <= y || mem x l')
```

utilité des foncteurs ? un exemple

Implémentation qui s'appuie sur la fonctions de comparaison polymorphe "magique" :

```
val (<=) : 'a -> 'a -> bool
```

C'est mal !

utilité des foncteurs ? un exemple

Implémentation qui s'appuie sur la fonctions de comparaison polymorphe "magique" :

```
val (<=) : 'a -> 'a -> bool
```

C'est mal !

S'appuie sur la *représentation mémoire* des valeurs, et ne correspond pas toujours à la notion attendue

Par exemple : files implémentées avec deux listes

`([3;2;1], [])` et `([], [1;2;3])` représentent la même file, mais ont une représentation mémoire différente

exemple : deuxième tentative

L'utilisateur doit pouvoir spécifier comment comparer les éléments à stocker dans l'ensemble

Essai n°2

exemple : deuxième tentative

L'utilisateur doit pouvoir spécifier comment comparer les éléments à stocker dans l'ensemble

Essai n°2 : on passe la fonction de comparaison en argument

```
type 'a t
val empty : 'a t
val add : ('a -> 'a -> bool) -> 'a -> 'a t -> 'a t
val mem : ('a -> 'a -> bool) -> 'a -> 'a t -> bool
```

exemple : deuxième tentative

L'utilisateur doit pouvoir spécifier comment comparer les éléments à stocker dans l'ensemble

Essai n°2 : on passe la fonction de comparaison en argument

```
type 'a t
val empty : 'a t
val add : ('a -> 'a -> bool) -> 'a -> 'a t -> 'a t
val mem : ('a -> 'a -> bool) -> 'a -> 'a t -> bool
```

Problème : rien n'empêche de changer de fonction de comparaison en cours de route.

Utilisation erronée mais permise par le système de types.

exemple : troisième tentative

Essai n°3

exemple : troisième tentative

Essai n°3 : on fixe la fonction de comparaison à la création de l'ensemble

```
type 'a t
val empty : ('a -> 'a -> int) -> 'a t
val add : 'a -> 'a t -> 'a t
val mem : 'a -> 'a t -> bool
```

```
type 'a t = { compare : 'a -> 'a -> bool; data : 'a list }
let empty cmp = { compare = cmp; data = [] }
...
```

exemple : troisième tentative

Problème : si on ajoute une fonction d'union d'ensembles :

```
val union : 'a t -> 'a t -> 'a t
```

alors rien n'interdit d'appeler `union` sur des ensembles construits avec des fonctions de comparaison différentes...

la bonne solution : un foncteur

```
module MakeSet (C: COMPARABLE) = struct ... end
```

avec

```
module type COMPARABLE = sig
  type t
  val compare : t -> t -> bool
end
```

foncteur : le code

```
module MakeSet (C: COMPARABLE) = struct
  type t = C.t list
  let empty = []
  let add x l =
    ... C.compare ...
  let mem x l =
    ... C.compare ...
end
```

foncteur : l'interface

```
module MakeSet (C: COMPARABLE) : sig
  type t
  val empty : t
  val add : C.t -> t -> t
  val mem : C.t -> t -> bool
end
```

syntaxe équivalente :

```
module MakeSet : functor (C: COMPARABLE) -> sig
  type t
  val empty : t
  val add : C.t -> t -> t
  val mem : C.t -> t -> bool
end
```

foncteur : utilisation

```
module Int = struct
  type t = int
  let compare x y = x <= y
end

module IntSet = MakeSet(Int)
(* module IntSet : sig
     type t
     val empty : t
     val add : Int.t -> t -> t
     val mem : Int.t -> t -> bool
   end *)

let s : IntSet.t =
  IntSet.add 3 (IntSet.add 4 IntSet.empty)
```

multiples instanciations

Différentes instanciations du foncteur produisent des types **incompatibles**

```
module Int1 = struct type t = int  let compare = (<=) end
module Int2 = struct type t = int  let compare = (>=) end

module IntSet1 = MakeSet(Int1)
module IntSet2 = MakeSet(Int2)

let s = IntSet2.add 4 IntSet1.empty
(*                    ^^^^^^^^^^^^
Error: This expression has type IntSet1.t = MakeSet(Int1).t
       but an expression was expected of type IntSet2.t
*)
```

C'est le comportement attendu !

foncteurs et contraintes de signatures

Lorsque l'on écrit la signature du foncteur : peut-on donner un nom à la signature du résultat ?

```
module MakeSet (C: COMPARABLE) : sig
  type t
  val empty : t
  val add : C.t -> t -> t
  val mem : C.t -> t -> bool
end
```

foncteurs et contraintes de signatures

```
module type SET = sig
  type t
  type element
  val empty : t
  val add : element -> t -> t
  val men : element -> t -> bool
end
module MakeSet (C: COMPARABLE) : SET
```

Problème : le type `element` n'est pas relié au type du paramètre `C.t`...

```
module IntSet = MakeSet(Int)
let s = IntSet.add 3 IntSet.empty
(*                   ^
Error: This expression has type int but an expression was expected
       of type IntSet.element
*)
```

contraintes

Le problème est que `SET` spécifie le type des *éléments de l'ensemble* comme un type abstrait `element`.

On peut le lier au type du paramètre avec une **contrainte** :

```
module MakeSet (C: COMPARABLE) : (SET with type element = C.t)
(* module MakeSet : functor (C: COMPARABLE) -> sig
     type t
     type element = C.t
     val empty : t
     val add : element -> t -> t
     val mem : element -> t -> bool
   end *)
```

foncteurs : applications

1. **structures de données paramétrées par des types de données**
   - `Hashtbl.Make` : tables de hachage
   - `Set.Make` : ensembles finis codés par des arbres équilibrés
   - `Map.Make` : tables d'association codées par des arbres équilibrés

2. **algorithmes paramétrés par des structures de données**

   exemple : algorithme de recherche de plus court chemin

   bon exemple : la bibliothèque `ocamlgraph` [1]

1. `https://github.com/backtracking/ocamlgraph`

algorithme générique

```
module DijkstraShortestPath
  (G : sig
         type graph
         type vertex
         type successors :
           graph -> vertex -> (vertex * float) list
       end) :
  sig
    val shortest_path :
      G.graph -> G.vertex -> G.vertex ->
      G.vertex list * float
  end
```

exercice

écrire un foncteur pour calculer modulo `m`

```
module Modular (M: sig val m : int end) : sig
  type t
  val of_int : int -> t
  val add : t -> t -> t
  val sub : t -> t -> t
  val mul : t -> t -> t
  val to_int : t -> int
end
```

du bon usage des foncteurs

il faut résister à la tentation de tout généraliser

suggestion : écrire un foncteur si

- il s'agit d'une bibliothèque générique
  (dont on ne connaît pas les clients)
- on a soi-même au moins deux instances à en faire

compilation des modules & foncteurs

compilation des modules (stratégie 1)

Les informations de types sont effacées

Stratégie 1 : compilation des modules comme des enregistrements

module $\rightarrow$ enregistrement

foncteur $\rightarrow$ fonction entre enregistrements

```
module M = struct
  type t = 'a list
  let empty = []
  let cons x l = x :: l
end
```

$\rightsquigarrow$

```
let m = {
  empty = [];
  cons = (fun x l -> x :: l);
}
```

compilation des foncteurs (stratégie 1)

Stratégie 1 : foncteurs deviennent des fonctions entre enregistrements

```
module M (C: COMPARABLE) =
struct
  type t = C.t list
  let empty = []
  let add x l =
    match l with
    | y :: l' ->
      if C.compare x y ...
      ...
end
```

$\rightsquigarrow$

```
let m c = {
  empty = [];
  add = (fun x l ->
    match l with
    | y :: l' ->
      if c.compare x y ...
      ...
  );
}
```

compilation des modules (stratégie 2)

Stratégie 2 : défonctorisation, i.e. ≪ dépliage ≫ complet des modules et foncteurs à la compilation

```
module M = struct
  type t = 'a list
  let empty = []
  let cons x l = x :: l
end
```

$\rightsquigarrow$

```
let _M_empty = []
let _M_cons x l = x :: l
```

compilation des foncteurs (stratégie 2)

Stratégie 2 : le code d'un foncteur est dupliqué *à chaque application du foncteur*

```
module M (C: COMPARABLE) =
struct
  type t = C.t list
  let empty = []
  let add x l =
    match l with
    | y :: l' ->
      if C.compare x y ...
      ...
end
module X = M(Int)
```

$\rightsquigarrow$

```
let _X_M_Int_empty = []
let _X_M_Int_add x l =
  match l with
  | y :: l' ->
    if _Int_compare x y ...
    ...
```

stratégies de compilation : comparaison

Stratégie 1 :

- simple, et peut être appliquée fichier par fichier
- peut être complémentée par des optimisations qui déplient les modules/foncteurs dans certains cas

→ utilisée par OCaml(+flambda), SML/NJ (compilateur SML)

Stratégie 2 :

- génère du code très efficace (modules n'ont aucun coût à runtime)
- duplique du code (pour les foncteurs appliqués plusieurs fois)
- doit être appliquée au programme entier

→ utilisée par MLton (compilateur SML optimisant), Futhark

types fantômes

(“phantom types” en anglais)

types fantômes

technique exploitant l'interaction entre **abstraction** et **types paramétrés**

pas exclusif à OCaml : aussi en Haskell, Rust, Swift, ...

objectif : refléter certains invariants d'un module dans les *paramètres* d'un type abstrait

types fantômes

Formellement :

dans `foo.mli` :

```
type 'a t
```

dans `foo.ml` :

```
type 'a t = string
```

Le paramètre `'a` n'est pas utilisé dans la définition : c'est un **paramètre de type fantôme**

types fantômes : motivation

on veut définir un module de tableaux accessibles en lecture seule

types fantômes : motivation

on veut définir un module de tableaux accessibles en lecture seule

```
module type ROARRAY : sig
  type 'a t
  val make : int -> 'a -> 'a t
  val get : 'a t -> int -> 'a
  val freeze : 'a array -> 'a t
end
```

types fantômes : motivation

on veut définir un module de tableaux accessibles en lecture seule

```
module type ROARRAY : sig
  type 'a t
  val make : int -> 'a -> 'a t
  val get : 'a t -> int -> 'a
  val freeze : 'a array -> 'a t
end

module ROArray : ROARRAY = struct
  type 'a t = 'a array
  let make = Array.make
  let get = Array.get
  let freeze a = Array.copy a
end
```

types fantômes : motivation

Inconvénient : on a deux fonctions get incompatibles :

- `Array.get : 'a array -> int -> 'a`
- `ROArray.get : 'a ROArray.t -> int -> 'a`

qui ont pourtant la même implémentation (idem pour `Array.length`...)

Solution :

- un seul type pour représenter à la fois `array` et `ROArray.t`
- le caractère RO/RW devient un **paramètre** du type
- une seule fonction get

types fantômes : tableaux RO/RW

interface :

```
module type FreezableArray : sig
  type ro
  type rw
  type ('a, 'perm) t  (* 'perm sera ro ou rw *)

  val make : int -> 'a -> ('a, 'perm) t
  val get : ('a, 'perm) t -> int -> 'a
  val set : ('a, rw) t -> int -> 'a -> unit
  val freeze : ('a, 'perm) t -> ('a, ro) t
end
```

types fantômes : tableaux RO/RW

implémentation :

```
module Array : FreezableArray = struct
  type ro = RO
  type rw = RW
  type ('a, 'perm) t = 'a array

  let make = Array.make
  let get = Array.get
  let set = Array.set
  let freeze a = Array.copy a
end
```

l'implémentation est la même ! on a simplement raffiné les types

récapitulons

- types `ro` / `rw` utilisés comme **marqueurs**, pas pour parler de données du programme
- paramètre fantôme `'perm` utilisé pour matérialiser des invariants associés aux valeurs du type (≪ s'autorise-t-on à les modifier ? ≫)
- permet d'écrire des fonctions polymorphes qui s'appliquent quel que soit l'invariant associé au type
- à l'exécution, tout ceci est effacé ! types vérifiés statiquement, *pas de coût supplémentaire à l'exécution*

autre exemple similaire

Sanitisation de chaînes de caractères :

```
type 'a str
type clean
type dirty

(* on peut calculer la longueur de n'importe quelle chaîne *)
val length : 'a str -> int
(* on se méfie de l'utilisateur : les inputs sont "sales" *)
val read : unit -> dirty str
(* il faut les nettoyer... *)
val sanitize : dirty str -> clean str
(* ...avant de pouvoir les afficher *)
val write : clean str -> unit
```

autre exemple similaire (2)

implémentation :

```
type 'a str = string
type clean = Clean
type dirty = Dirty

let length = String.length
let read = read_line ()
let sanitize s = ...
let write s = Printf.printf "%s\n" s
```

dernier exemple : interaction avec les foncteurs

bibliothèque de vecteurs mathématiques :

```
type vec
(* [zero n]: vecteur nul de dimension n *)
val zero : int -> vec
(* [u k n]: k-ème vecteur unité de dimension n *)
val u : int -> int -> vec
val add : vec -> vec -> vec
val smul : float -> vec -> vec
```

dernier exemple : interaction avec les foncteurs

bibliothèque de vecteurs mathématiques :

```
type vec
(* [zero n]: vecteur nul de dimension n *)
val zero : int -> vec
(* [u k n]: k-ème vecteur unité de dimension n *)
val u : int -> int -> vec
val add : vec -> vec -> vec
val smul : float -> vec -> vec
```

Problème : on peut mélanger des vecteurs de tailles (= dimensions) différentes !

types fantômes et foncteurs

Introduisons un paramètre fantôme :

```
type 'dim vec
val add : 'dim vec -> 'dim vec -> 'dim vec
val smul : float -> 'dim vec -> 'dim vec
```

garantit que `add` et `smul` agissent sur des vecteurs de même dimension

problème restant :

```
val zero : int -> ?? vec
val u : int -> int -> ?? vec
```

Quel types pour le paramètre fantôme `'dim` ?

types fantômes et foncteurs

On peut utiliser un foncteur pour **générer** un nouveau type pour chaque dimension fournie

```
module type DIM = sig
  val dim : int
end

module Make (Dim: DIM) : sig
  type dim (* le marqueur fantôme pour la dimension Dim.dim *)
  val zero : unit -> dim vec
  val u : int -> dim vec
end
```

utilisation

```
module V2 = Make (struct let dim = 2 end)
(* module V2 : sig
     type dim
     val zero : unit -> dim vec
     val u : int -> dim vec
   end *)
module V3 = Make (struct let dim = 3 end)
(* module V3 : sig type dim ...[idem]... end *)

(* le vecteur (1, 1) *)
let v = add (V2.u 1.) (V2.u 2.)
(* le vecteur (3, 0, 1) *)
let v' = add (smul 3. (V3.u 1.)) (V3.u 3)
(* Erreur de type! *)
let v'' = add (V2.u 1) (V3.zero ())
```

pour récapituler

ingrédients clés :

- L'**abstraction** pour cacher des détails d'implémentation
- Les **foncteurs** pour écrire du code générique

types fantômes : technique d'abstraction « avancée » combinant types abstraits et polymorphisme