Les objets et les classes d'Objective Caml

Généralités sur les objets

Un objet regroupe ensemble

- Un état interne:
  - une ou plusieurs variables d'instance;
  - accessibles seulement aux méthodes de l'objet;
  - mutables ou non.
- Une collection de méthodes:
  - des fonctions qui opèrent sur l'état interne;
  - accessibles de l'extérieur;
  - assurent l'interface entre l'extérieur et l'état interne.

La seule opération sur un objet est l'envoi de méthode, noté

objet#*méthode* [*arguments*]

Exemple: un véhicule à moteur

État interne:

- Position de l'accélérateur
- Position de la pédale de freins
- Rapport de la boîte de vitesses
- Régime du moteur

Méthodes:

- Positionner accélérateur, freins
- Changer de rapport
- Calculer la vitesse

Classes et objets

Une classe est un moule à objets:

- Elle définit les méthodes.
- Elle déclare les variables d'instances et définit leur valeur initiale.
- Elle prend en argument un ou plusieurs paramètres d'initialisation qui servent à déterminer les valeurs initiales des variables d'instance.

La construction `new` renvoie un nouvel objet – une nouvelle instance de la classe:

`new` *nom-de-classe paramètres initiaux*

```
class moto () =
  val mutable rapport = 0

  method passer =
    if rapport > 6 then failwith "passer" else rapport <- rapport + 1

  method rentrer =
    if rapport <= 0 then failwith "rentrer" else rapport <- rapport - 1

  method rapport_courant = rapport
end

let m = new moto ()
     val m : moto = <obj>

m#passer; m#passer; m#rentrer; m#rapport_courant
     - : int = 1
```

Exemple long: les entrées-sorties unifiées

Une classe qui encapsule les impressions sur le terminal:

```
class standard_output () =
  method put_char c = print_char c
  method put_string s = print_string s
  method put_int n = print_int n
end

let stdout = new standard_output ()
     val stdout : standard_output = <obj>

stdout#put_string "Hello!\n"
     Hello!
     - : unit = ()
```

Polymorphisme sur les objets

```
let euros obj francs =
  obj#put_int francs;
  obj#put_string " francs = ";
  obj#put_int (francs * 100 / 655);
  obj#put_string " euros\n"

     val euros :
     < put_int : int -> 'a; put_string : string -> 'b; .. > -> int -> 'b
```

`euros` s'applique à n'importe quel objet possédant au moins les méthodes `put_int` et `put_string` avec les types indiqués.

```
euros stdout 322;;
     322 francs = 49 euros
     - : unit = ()
```

Quel est le type de `stdout`?

La définition de la classe **`standard_output`** définit également une abréviation de type pour le type des objets de cette classe:

```
type standard_output =
    < put_char : char -> unit;
      put_string : string -> unit;
      put_int : int -> unit >
```

Sorties sur fichiers

```
class fileout filename =
  val chan = open_out filename
  method put_char c = output_char chan c
  method put_string s = output_string chan s
  method put_int n = output_string chan (string_of_int n)
  method seek pos = seek_out chan pos
  method close = close_out chan
end
```

En plus des opérations `put_` vues précédemment, on ajoute la possibilité de positionnement à l'intérieur du fichier (`seek`) et la fermeture explicite (`close`).

Utilisation de la classe `fileout`:

```
let f = new fileout "/tmp/myfile";;
f#put_string "Hello, world!\n";;
euros f 1234;;
f#seek 7;;
f#put_string "monde";;
f#close;;
```

L'appel `euros f 1234` est bien typé car `f` fournit les méthodes `put_string` et `put_int` requises (et d'autres).

Implémentation de méthodes par défaut

Lorsqu'on sait faire `put_char`, on sait également implémenter `put_string` (imprimer chaque caractère) et `put_int` (convertir l'entier en chaîne).

Cependant, sur certains types de sorties, on dispose d'implémentations plus efficaces (p.ex. `put_string` sur un fichier, où l'on peut faire des écritures par blocs).

- Utilisation d'une classe virtuelle pour définir les implémentations par défaut une fois pour toute.
- Utilisation de l'héritage entre classes pour récupérer les implémentations par défaut dans une classe de sortie particulière.
- Redéfinition dans la classe de sortie des méthodes pour lesquelles on a une meilleure implémentation.

La classe virtuelle des “écrivains”

```
class virtual writer () as self =
  virtual put_char : char -> unit
  method put_string s =
    for i = 0 to String.length s - 1 do self#put_char s.[i] done
  method put_int n =
    self#put_string (string_of_int n)
end
```

Cette classe est marquée “virtuelle” car certaines de ses méthodes ne sont pas définies (`put_char`); on ne peut donc pas créer d’instances de cette classe.

L’identificateur `self` est lié (au moment de l’appel de méthode) à l’objet lui-même. Il permet donc à une méthode de rappeler une autre méthode du même objet.

De `writer` vers `fileout` par héritage

Une implémentation simple des sorties sur fichiers hérite **`put_string`** et **`put_int`** de **`writer`**, et définit uniquement les autres méthodes:

```
class fileout filename =
  inherit writer()
  val chan = open_out filename
  method put_char c = output_char chan c
  method seek pos = seek_out chan pos
  method close = close_out chan
end
```

Cette classe se comporte exactement comme la classe **`fileout`** vue précédemment (implémentée entièrement à la main):

```
let f = new fileout "/tmp/myfile";;
f#put_string "Hello, world!\n";;
f#close;;
```

De `writer` vers `fileout`, optimisé

Dans la classe **`fileout`**, nous pouvons fournir une implémentation plus efficace de **`put_string`**, qui va masquer l'implémentation par défaut héritée de **`writer`**:

```
class fileout filename =
  inherit writer()
  val chan = open_out filename
  method put_char c = output_char chan c
  method put_string s = output_string chan s
  method seek pos = seek_out chan pos
  method close = close_out chan
end
```

Nous conservons l'implémentation par défaut de **`put_int`**, car elle est suffisamment efficace.

La liaison tardive (récursion ouverte)

Les liaisons entre méthodes sont déterminées au moment où l'on fait **`new`**, et non pas à la définition des classes. C'est ce qu'on appelle la liaison tardive, par contraste avec la liaison statique en usage dans le reste du langage.

```
class c () as self =
  method a = print_string "c"          let a() = print_string "c"
  method b = self#a                    let b() = a()
end

class d () =
  inherit c ()
  method a = print_string "d"          let a() = print_string "d"
end

let o = new d()

o#b                                    b()
(* imprime "d" *)                       (* imprime "c" *)
```

Méthodes `protected`

Une méthode marquée `protected` ne peut être appelée directement de l'extérieur d'un objet, mais seulement depuis une autre méthode de l'objet. C'est utile pour définir des opérations locales à une classe.

```
class stringout () as self =
  inherit writer()
  val mutable accu = String.create 20
  val mutable pos = 0
  method protected redim =
    let newsize = String.length accu * 2 in
    let newaccu = String.create newsize in
    String.blit accu 0 newaccu 0 (String.length accu);
    accu <- newaccu
  method put_char c =
    if pos >= String.length accu then self#redim;
    accu.[pos] <- c;
    pos <- pos + 1
  method contents = String.sub accu 0 pos
end
```

Sous-typage sur les objets

Si $A : \langle m_1 : \tau_1;\ \ldots;\ m_k : \tau_k\rangle$
et $B : \langle m_1 : \tau_1;\ \ldots;\ m_k : \tau_k;\ m_{k+1} : \tau_{k+1};\ \ldots\rangle$
alors B peut être utilisé partout à la place de A.

On dit que le type de B est *sous-type* du type de A
(cf. la notion de sous-ensemble).

Application: on peut mettre des objets de différent types dans une structure de donnée homogène (p.ex. liste, tableau), à condition de voir tous ces objets comme appartenant à leur super-type commun.

Sous-typage en Objective Caml

```
let w1 = new fileout "/tmp/myfile" in
let w2 = new stringout () in

let writers = [(w1 :> writer); (w2 :> writer)] in

List.iter (fun w -> w#put_string "Hello") writers;
w1#close;
w2#contents
```

En Objective Caml, le sous-typage doit être marqué explicitement comme une coercion de types:

(*objet* :> *type*)

(*objet* : *type-source* :> *type-résultat*)

Polymorphisme vs. sous-typage

Le polymorphisme permet d'avoir des fonctions s'appliquant à des arguments de différents types, mais pas des structures de données hétérogènes:

`List.rev [1;2];;`	*- : int list = [2;1]*
`List.rev [true;false];;`	*- : bool list = [false;true]*
`[ [1]; [true] ];;`	*Erreur de type*

`int list` et `bool list` n'ont pas de supertype commun.

Seul le sous-typage permet les structures de données hétérogènes.

Faire référence à la superclasse

Après les sorties génériques, faisons maintenant les entrées sur le même moule:

```
class virtual reader () as self =
  virtual get_char : char
  method get_string n =
    let s = String.create n in
    for i = 0 to n - 1 do s.[i] <- self#get_char done;
    s
  method get_int =
    let rec read accu =
      match self#get_char with
        '0'..'9' as c -> read (10 * accu + Char.code c - Char.code '0')
      | _ -> accu
    in read 0
end
```

Problème: `get_int` lit un caractère de trop!

Faire référence à la superclasse, suite

C'est facile à corriger si l'on sait faire **`seek`** sur l'entrée: on revient en arrière d'un caractère après avoir lu l'entier.

```
class virtual seekable_reader () as self =
  inherit reader() as super
  virtual seek : int -> unit
  virtual pos : int             (* position courante de lecture *)
  method get_int =
    let res = super#get_int in
    self#seek (self#pos - 1);
    res
end
```

Le `as super` donne le nom `super` aux méthodes de la superclasse. `super#get_int` fait alors référence à l'implémentation de `get_int` dans la superclasse. On peut ainsi redéfinir une méthode héritée tout en réutilisant l'implémentation héritée.

Lecture sur fichier

```
class filein filename =
  inherit seekable_reader()
  val chan = open_in filename
  method get_char = input_char chan
  method seek pos = seek_in chan pos
  method pos = pos_in chan
  method close = close_in chan
end
```

Héritage multiple

Une classe peut hériter de plusieurs autres classes.

```
class pipe () =
  val q = Queue.create ()
  inherit writer()
  inherit reader()
  method put_char c = Queue.add c q
  method get_char =
    try Queue.take q
    with Queue.Empty -> raise End_of_file
end
```

Une instance de `pipe` est utilisable comme écrivain aussi bien que comme lecteur:

```
let p = new pipe();;
euros p 100;;
p#get_string 13;;             - : string = "100 francs = "
p#get_int;;                   - : int = 15
```

Sous-typage et héritage

Le sous-typage:

- permet de considérer des implémentations différentes avec la même interface;
- autorise les fonctions génériques;
- autorise le mélange de différentes classes dans une même structure de données.

L'héritage:

- permet la programmation incrémentale et la réutilisation de code;
- apporte la liaison tardive.

Contrairement à la plupart des langages à objets, OCaml dissocie complètement sous-typage et héritage:

- Une classe A peut être sous-type d'une classe B sans hériter de B.
  (Implémentations indépendantes d'une même interface, ou d'interfaces en relation de sous-typage.)

- Une classe A peut hériter de B sans être sous-type de B.
  (Cas rare, se produit avec selftype $+$ méthodes binaires.)

Classes paramétrées

Tout comme les types concrets, les classes peuvent être paramétrées par un ou plusieurs types, ce qui leur ajoute du polymorphisme.

Exemple typique: les classes “conteneurs”.

```
class ('a, 'b) hashtable size =
  val buckets = (Array.make size [] : ('a * 'b) list array)
  method find key =
    let h = (Hashtbl.hash key) mod (Array.length buckets) in
    List.assoc key buckets.(h)
  method add key data =
    let h = (Hashtbl.hash key) mod (Array.length buckets) in
    buckets.(h) <- (key, data) :: buckets.(h)
end
```

Les types concrets à la sauce objet

```
class virtual 'a list () as self =
  virtual isnil : bool
  virtual head : 'a
  virtual tail : 'a list
  method length =
    if self#isnil then 0 else 1 + self#tail#length
end
```

```
class 'a nil () =
  inherit ('a) list ()
  method isnil = true
  method head = failwith "nil"
  method tail = failwith "nil"
end
```

```
class 'a cons h0 t0 =
  val h = h0
  val t = t0
  inherit ('a) list ()
  method isnil = false
  method head = h
  method tail = t
end
```

Codage populaire en Java, mais peu utile en Caml car les types concrets du langage sont généralement plus commodes.

Des tables de hachage encore plus orientées objet

Les clés sont des objets qui portent leur propres méthodes **`hash`** et **`equal`**:

```
class ('a, 'b) hashtable size =
  val buckets = (Array.make size [] : ('a * 'b) list array)
  method find (key : 'a) =
    let h = key#hash mod (Array.length buckets) in
    let rec find = function
        [] -> raise Not_found
      | (k, d) :: rest -> if key#equal k then d else find rest
    in find buckets.(h)
  method add key data =
    let h = key#hash mod (Array.length buckets) in
    buckets.(h) <- (key, data) :: buckets.(h)
end
```

La réponse du système fait apparaître que le premier argument du constructeur de types **hashtbl** doit être un type objet avec au moins les méthodes **hash** et **equal**:

```
class ('a, 'b) hashtable (int) =
  constraint 'a = < hash : int; equal : 'a -> bool; .. >
  val buckets : ('a * 'b) list array
  method add : 'a -> 'b -> unit
  method find : 'a -> 'b
end
```

C'est une erreur d'utiliser **hashtbl** autrement:

```
type int_map = (int, int) hashtable
     This type parameter int should be an instance of type
        < hash : int; equal : 'a -> bool; .. > as 'a
```

Les modules d'Objective Caml

La programmation modulaire

- Découpage du programme en morceaux compilables indépendamment

  Rendre les gros programmes compilables

- Donner de la structure au programme et faciliter la réutilisation de certaines parties

  Rendre les gros programmes compréhensibles

Les modules de ML

- Un petit langage fonctionnel typé.
- Manipule des collections de définitions (de valeurs, de types) du langage de base.
- Leurs types: des collections de déclarations/spécifications (types des valeurs, déclarations des types).
- Modules emboîtés (en guise de structure de données).
- Fonctions (foncteurs) et application de fonctions.

Largement indépendant du langage de base.

Modules de base

Les *structures*: collections de définitions.

`struct` *définition* ... *définition* **`end`**

Les définitions correspondent aux phrases du langage de base:

définitions de valeurs et de fonctions	**`let`** $x = \ldots$
définitions de types	**`type`** $t = \ldots$
définitions d'exceptions	**`exception`** E [**`of`** ...]
définitions de classes	**`class`** $C = \ldots$ **`end`**
définitions de sous-modules	**`module`** $X = \ldots$
définitions de type de module	**`module type`** $S = \ldots$

Nommage d'un module

On donne un nom à un module à l'aide de la liaison `module`.

```
module S = struct ... end
```

Exemple:

```
module S =
  struct
    type t = int
    let x = 0
    let f x = x+1
  end
```

Utilisation d’un module

On fait référence aux composantes d’un modules avec la notation pointée *module*.*composante*

```
... (S.f S.x : S.t) ...
```

Autre possibilité: la directive `open` *module* permet d’omettre le préfixe et le point:

```
open S
... (f x : t) ...
```

Modules emboîtés

Un module peut être composante d'un autre module:

```
module T =
  struct
    module R = struct let x = 0 end
    let y = R.x + 1
  end
```

La notation pointée et le `open` s'étendent naturellement:

```
T.R.x
```

```
open T.R
```

Les types des modules de base

Les *signatures*: collections de spécifications (de types).

sig *spécification* ... *spécification* **end**

spécification de valeurs	**val** $x : \sigma$
spécification de types, abstraites	**type** t
spécification de types, manifestes	**type** $t = \tau$
spécification d'exceptions	**exception** E
spécification de classes	**class** $C = \ldots$ **end**
spécification de sous-modules	**module** $X : M$
spécification de type de module	**module type** $S\ [= M]$

Nommage d'un type de module: par la liaison `module type`

```
module type SIG = sig ... end
```

Restriction par une signature

La construction (*structure* : *signature*)

- vérifie que la structure satisfait la signature
  (toutes les composantes spécifiées dans la signatures doivent être définies dans la structure, avec des types au moins aussi généraux);

- rend inaccessibles les parties de la structure qui ne sont pas spécifiées dans la signature;

- produit un résultat qui peut être lié par **`module`** `M = ...`

Sucre syntaxique:

`module` `M :` *signature* = *structure*

est équivalent à

`module` `M = (`*structure* : *signature*)

Exemple de restriction

```
module S =
  (struct
      type t = int
      let x = 1
      let y = x + 1
   end :
   sig
      type t
      val y : t
   end)
```

`S.x;;` *erreur ''S.x is unbound''*

`S.y + 1;;` *erreur ''S.y is not of type int''*

Vues multiples d'un même module

```
module Stamp =
  struct
    type t = int
    let new () = ...
    let equal s1 s2 = (s1 = s2)
  end

module type ABSTRSTAMP =
  sig
    type t
    val equal: t -> t -> bool
  end

module AbstrStamp = (Stamp : ABSTRSTAMP)
```

Les utilisateurs de `AbstrStamp` peuvent seulement tester l'égalité de deux stamps; ceux de `Stamp` peuvent aussi créer de nouveaux stamps et les traiter comme des entiers (affichage).

Pas de duplication de code entre `AbstrStamp` et `Stamp`.

Modules et compilation séparée

Une unité de compilation A se compose de deux fichiers:

- Le fichier d'implémentation A.`ml`:
  une suite de définitions
  semblable à l'intérieur de `struct...end`

- Le fichier d'interface A.`mli` (optionnel):
  une suite de spécifications
  semblable à l'intérieur de `sig...end`

Une autre unité de compilation B peut faire référence à A comme si c'était une structure, en utilisant la notation pointée A.`x` ou bien en faisant `open` A.

Compilation séparée d'un programme

Fichiers sources: `a.ml`, `a.mli`, `b.ml`

Étapes de compilation:

```
ocamlc -c a.mli                 (compile l'interface de A, crée a.cmi)
ocamlc -c a.ml                  (compile l'implémentation de A, crée a.cmo)
ocamlc -c b.ml                  (compile l'implémentation de B, crée b.cmo)
ocamlc -o monprog a.cmo b.cmo   (édition de liens finale)
```

Le programme se comporte comme le code monolithique suivant:

`module A = (struct` *contenu de a.ml* `end : sig` *contenu de a.mli* `end)`
`module B = struct` *contenu de b.ml* `end`

L'ordre des définitions de modules correspond à l'ordre des fichiers objets `.cmo` sur la ligne de commande de l'éditeur de liens.

Modules paramétrés

Un *foncteur* est une fonction des modules dans les modules:

`functor(`*S*`:` *signature*`) ->` *module*

Le *module* (corps du foncteur) est explicitement paramétré par le paramètre de module *S*. Il fait référence aux composantes de son paramètre avec la notation pointée.

```
module T = functor(S : SIG) ->
  struct
    type u = S.t * S.t
    let y = S.g(S.x)
  end
```

Application de foncteur

On ne peut pas directement accéder dans `T`. Il faut au préalable l'appliquer explicitement à une implémentation de la signature `SIG` (tout comme on applique une fonction normale).

```
module T1 = T(S1)
module T2 = T(S2)
```

`T1`, `T2` s'utilisent alors comme des structures normales:

```
(T1.y : T2.u)
```

`T1` et `T2` partagent entièrement leur code.

Exemple long: les polynômes sur un anneau quelconque

On va implémenter les opérations usuelles sur les polynômes à coefficients dans un anneau quelconque, donné en paramètre.

Le paramètre anneau est une structure ayant la signature suivante:

```
module type ANNEAU =
  sig
    type t               (* type des éléments de l'anneau *)
    val zero: t
    val un: t
    val plus: t->t->t    (* somme *)
    val opp: t->t        (* opposé *)
    val mult: t->t->t    (* produit *)
  end
```

Polynômes pleins

```
module PolyPlein = functor (Anneau : ANNEAU) ->
  struct
    module A = Anneau
    type t = A.t array                   (* le type des polynômes *)
    let zero = [| A.zero |]              (* le polynôme nul *)
    let un = [| A.zero; A.un |]          (* le polynôme X *)
    let monôme c n =                     (* le monôme c.X^n *)
      let p = Array.create (n+1) A.zero in
      p.(n) <- c; p
    let coeff n p = p.(n)                (* renvoie le coeff de X^n dans p *)
    let degré p =                        (* renvoie le degré de p *)
      let d = ref (Array.length p - 1) in
      while !d >= 0 && p.(!d) = A.zero do decr d done;
      !d
    let plus p1 p2 =                     (* addition *)
      let r = Array.create (max (Array.length p1) (Array.length p2)) A.zero in
      for i = 0 to Array.length p1 - 1 do r.(i) <- p1.(i) done;
      for i = 0 to Array.length p2 - 1 do r.(i) <- A.plus r.(i) p2.(i) done;
      r
    let opp p = ...                      (* opposé *)
    let mult p1 p2 = ...                 (* multiplication *)
  end
```

Utilisation du foncteur des polynômes

Polynômes à coefficients entiers:

```
module Entiers =
  struct
    type t = int
    let zero = 0
    let un = 1
    let plus x y = x + y
    let opp x = -x
    let mult x y = x * y
  end
module PolyEntiers = PolyPlein(Entiers)
```

Polynômes à coefficients entiers à deux variables = polynômes de polynômes d'entiers:

```
module Poly2Entiers = PolyPlein(PolyEntiers)
```

C'est correct car la structure résultat de `PolyPlein` satisfait la signature `ANNEAU`.

Polynômes creux

```
module PolyCreux = functor (Anneau : ANNEAU) ->
  struct
    module A = Anneau
    type t = (A.t * int) list           (* le type des polynômes *)
    let zero = []                       (* le polynôme nul *)
    let un = [(A.un, 1)]                (* le polynôme X *)
    let monôme c n = [(c, n)]           (* le monôme c.X^n *)
    let rec degré = function
        [] -> 0
      | [(coeff, d)] -> d
      | (coeff, d) :: reste -> degre reste
    let rec coeff n p =
      match p with
        [] -> A.zero
      | (c, d) :: reste ->
          if d = n then c else
          if d > n then A.zero else coeff n reste
    let plus p1 p2 =  ...               (* addition *)
    let opp p = ...                     (* opposé *)
    let mult p1 p2 = ...                (* multiplication *)
  end
```

Indépendance vis-à-vis de la représentation

`PolyPlein` et `PolyCreux` implémentent la même interface et sont essentiellement interchangeables:

```
module PolyEntiers = PolyCreux(Entiers)
module Poly2Entiers = PolyCreux(PolyEntiers)
```

Mais pas totalement interchangeables: le type de représentation des polynômes reste apparent.

```
module P1 = PolyPlein(Entiers);;        (* P1.t = int array *)
P1.degré [| 1;2;3;4 |];;                (* - : int = 3 *)

module P2 = PolyCreux(Entiers);;        (* P2.t = (int * int) list *)
P2.degré [| 1;2;3;4 |];;                (* erreur de type *)
```

Cacher la représentation des polynômes

On applique une contrainte de signature au résultat des foncteurs `PolyPlein` et `PolyCreux` pour rendre abstrait le type `t`.

Premier essai:

```
module type POLYNÔME =
  sig
    module A : ANNEAU
    type t
    val zero: t                              val un: t
    val monôme: A.t -> int -> t              val degré: t -> int
    val coeff: int -> t -> A.t               val opp: t -> t
    val plus: t -> t -> t                    val mult: t -> t -> t
  end
module PolyPlein = functor (Anneau: ANNEAU) -> (struct ... end : POLYNÔME)
```

Problème: on a aussi rendu abstrait le type `A.t` des coefficients, ce qui rend le résultat de `PolyPlein` inutilisable:

```
module PolyEntiers = PolyPlein(Entiers);;
PolyEntiers.monôme 15 2;;
```

This expression has type int but is used with type PolyEntiers.A.t

La bonne solution: utiliser un type manifeste pour conserver l'identité du type `A.t` du résultat:

```
module PolyPlein = functor (Anneau: ANNEAU) ->
  (struct ... end : POLYNÔME with type A.t = Anneau.t)
```

Dès lors:

```
module PolyEntiers = PolyPlein(Entiers)
```

On a `PolyEntiers.t` abstrait,
mais `PolyEntiers.A.t` = `Entiers.t` = `int` comme désiré.

La notation `with`

Permet d'ajouter des égalités de types dans une signature existante.

L'expression `POLYNÔME with type A.t = Anneau.t` est une abréviation pour la signature

```
sig
  module A : sig type t = Anneau.t
                 val zero: t   val un: t
                 val plus: t->t->t
                 val opp: t->t
                 val mult: t->t->t  end
  type t
  val zero: t
  val un: t
  val monôme: A.t -> int -> t
  val plus: t -> t -> t
  val opp: t -> t
  val mult: t -> t -> t
  val degré: t -> int
  val coeff: int -> t -> A.t
end
```

Modules vs. polymorphisme

Les polynômes s'expriment aussi sans foncteurs, avec juste du polymorphisme à la ML:

```
type 'a polynôme
type 'a anneau =
  { zero: 'a; un: 'a; plus: 'a->'a->'a; opp: 'a->'a; mult: 'a->'a->'a }
val zero: 'a anneau -> 'a polynôme
val un: 'a anneau -> 'a polynôme
val monôme: 'a anneau -> 'a -> int -> 'a polynôme
val plus: 'a anneau -> 'a polynôme -> 'a polynôme -> 'a polynôme
val opp: 'a anneau -> 'a polynôme -> 'a polynôme
val mult: 'a anneau -> 'a polynôme -> 'a polynôme -> 'a polynôme
val degré: 'a anneau -> 'a polynôme -> int
val coeff: 'a anneau -> int -> 'a polynôme -> 'a
```

Premier problème: il faut passer les opérations de l'anneau en argument supplémentaire aux opérations sur les polynômes.

Foncteurs = paramétrisation en masse

Langage de base = paramétrisation individuelle

Deuxième problème: le type des polynômes ne reflète que le type du support de l'anneau, pas l'anneau lui-même.

```
let entiers =
  { zero = 0; un = 1; plus = fun x y -> x + y; ...}
let z_sur_3z =
  { zero = 0; un = 1; plus = fun x y -> (x + y) mod 3; ...}
let p1 = monôme entiers 15 2
let p2 = monôme z_sur_3z 2 3
let p3 = mult entiers p1 p2
```

Il est absurde de mélanger `p1` et `p2`, et pourtant ils ont le même type `int polynôme`.

La formulation à base de foncteurs détecte cette erreur par typage statique:

```
module Z_sur_3Z =
  struct let zero = 0  let un = 1  let plus x y = (x+y) mod 3 ... end
module PolyEntiers = PolyPlein(Entiers)
module PolyZ3Z = PolyPlein(Z_sur_3Z)
```

Les types `PolyEntiers.t` et `PolyZ3Z.t` sont tous deux abstraits, donc a priori incompatibles:

```
PolyZ3Z.degré PolyEntiers.un
```

This expression has type PolyEntiers.t
but is here used with type PolyZ3Z.t

Le polymorphisme du langage ML de base:

$$\begin{array}{rlcl} \text{types paramétrés:} & \text{type} & \longrightarrow & \text{type} \\[2ex] \text{fonctions:} & \text{valeurs} & \longrightarrow & \text{valeurs} \end{array}$$

Le langage de modules:

$$\begin{array}{rlcl} & \text{types} & & \text{types} \\ \text{foncteurs:} & \text{interprétés par} & \longrightarrow & \text{interprétés par} \\ & \text{des valeurs} & & \text{des valeurs} \end{array}$$

Cas particulier: $\mathbf{Z}/2\mathbf{Z}$

Autre point fort de la solution à base de foncteurs: on peut fournir des implémentations directes (sans passer par le foncteur) pour certains types.

Exemple: pour implémenter efficacement $\mathbf{Z}/2\mathbf{Z}[X]$, on fournit une structure `Z2Z` à base de tableaux de bits avec la même interface que le résultat du foncteur `PolyPlein`:

```
module Poly_Z2Z : (POLYNÔME with type A.t = int) =
  struct
    module A = Z_sur_2Z
    type t = bitvect
    let add = bitvect_xor
    let mult = bitvect_and
    let op = bitvect_neg
    ...
  end
```

Le reste du programme ne change pas.

Modules vs. objets

Un certain recouvrement dans le cas des types abstraits modifiables en place et avec opérations unaires:

```
module Hashtbl:
  sig
    type ('a, 'b) t
    val empty: int -> ('a, 'b) t
    val add: ('a, 'b) t -> 'a -> 'b -> unit
    val find: ('a, 'b) t -> 'a -> 'b
  end

class ('a, 'b) hashtbl (int) =
  method add: 'a -> 'b -> unit
  method find: 'a -> 'b
end
```

Foncteurs d'extension = un cas simple d'héritage.

Modules:

- Opérations n-aires.
- Plusieurs types abstraits reliés.
- Foncteurs généraux.

Objets:

- Sous-typage entre valeurs.
- Liaison tardive (récursion ouverte).

Syntaxe des structures et des signatures

```
Structures (implémentations)          Signatures  (interfaces)
------------------------------------------------------------------------
Séquence de définitions               Séquence de spécifications de types
```

Valeurs

```
let x = [1]                            val x : int list
let rec f x = ...                      val f : int -> int
```

Exceptions

```
exception E of string                  exception E of string
```

Types concrets

```
type t = int                           type t = int
type 'a u = A | B of 'a                type 'a u = A | B of 'a
```

Types abstraits

```
type v = string                        type v
```

Syntaxe des structures et des signatures (traits avancés)

```
  Structures                                 Signatures
-------------------------------------------------------------------------
```

Classes

```
class c x0 =                                class c (int) :
  val x = x0                                  val x: int
  method get = x                              method get: int
end                                         end
```

Sous-modules

```
module X = struct ... end                   module X : sig ... end
```

Types de modules, concrets

```
module type T = sig ... end                 module type T = sig ... end
```

Types de modules, abstraits

```
module type U = sig ... end                 module type U
```