Classes et modules

Didier Rémy
2001 - 2002

http://cristal.inria.fr/~remy/mot/4/
http://www.enseignement.polytechnique.fr/profs/informatique/I

Cours	
1. Point de rencontre 2. Groupement des classes 3. Constructeurs de classes 4. Abstraction et héritage 5. Classes amies 6. Combinaison intime	1. Structures de d (Listes, concat lules) 2. Bank 3. Ensembles 4. Polynomes

Résumé

Modules et classes remplissent des rôles différents.

- Les modules gèrent l'abstraction de type et de valeur e
  paramétrisation par des types et des valeurs *en gros*.
- Les classes n'offrent pas d'abstraction de type, ni la po
  groupement.
- Elles offrent le mécanisme de liaison tardive (classes), l
  (classe) et bien sûr la programmation par envoi de mess

Il est fréquent de combiner les deux mécanismes pour bén
les atouts simultanément.

Point de rencontre des classes et des n

Quelques exemples s'expriment aussi facilement dans le la
classes que dans celui des modules.

L'intersection est à la fois grande et petite :

- De nombreux programmes peuvent être écrits dans un
  l'autre.
- Cependant, la modularité n'est pas toujours la même s
  utilisé.
- Pour des exemples peu modulaire et qui utilisent peu le
  objet, le choix du style n'est pas fondamentale... au poi
  exemples peuvent également être écrit dans le langage d

Point de rencontre : types abstraits algé

... avec modification en place et des opérations unaires !

```
class ['a] stack = object

  val mutable p : 'a list = []
  method push v = p <- v :: p
  method pop =
    match p with
    | h :: t -> p <- t; h
    | [] -> failwith "Empty"
end;;
```

```
module Stack : STA
  type 'a t = 'a l
  let create () =
  let push x s = s
  let pop s =
    match !s with
      h::t -> s :=
    | [] -> failwit
end;;
```

Résumé	**classe**	**mod**
Le type des piles	`'a stack`	`'a S`
Création d'une pile	`new stack`	`Stac`
Empiler x sur s	s`#push` x	`Stac`
Dépiler s	s`#pop`	`Stac`

Extensibilité

```
class ['a] stack_ext =
  object (self)
    inherit ['a] stack




    method top =
      let s = self#pop in
      self#push s; s
end;;
```

```
module StackExt =
  (* include Stack
  type 'a t = 'a S
  let create = Sta
  let push = Stack
  let pop = Stack.
  let top p =
    let s = pop p
    push s p; s
end;;
```

Cependant, les modules ne permettent pas la liaison tardiv
les fonctions sont définies récursivement). Par exemple, la
`pop` n'affectera pas le comportement de `top`.

Quelle approche choisir ?

C'est une affaire de goût quand les deux sont possibles

Classes quand la liaison tardive est nécessaire.

Modules s'il y a des méthodes binaires or deux types abs
vecteurs et matrices (leurs représentations doivent être pa

Approche combinée dans de nombreux cas.

- utilisation orthogonale : chacun son rôle, indépendemm
- combinaison intime : l'un complète l'autre.

Regroupement les classes liées

C'est une utilisation orthogonale de modules et des classes

Les classes `nil` et `cons` sont liées, mais seulement par l'us

```
module Liste = struct
  exception Nil
  class ['a] nil =
    object (self : 'alist)
      method hd : 'a = raise Nil
      method tl : 'alist = raise Nil
      method null = true
    end
```

```
  class ['a] cons
    object (_ : '
      val hd = h
      method hd :
      method tl :
      method null
    end
end;;
```

NOTE : En Ocaml, les listes sont plus naturellement repr
type somme qui permet en outre d'utiliser le filtrage.

Extensibilité de classes liées

Typiquement, les deux classes seront étendues simultaném
cela ne soit pas obligatoire)

```
module Liste1 = struct
  class ['a] nil =
    object
      inherit ['a] Liste.nil
      method length = 0
    end
      ;;
```

```
class ['a] cons h t
  object
    inherit ['a] Lis
    method length =
      1+tl#length
  end
end;;
```

On peut aussi étendre les listes par l'ajout d'un nouveau c
`append`, ce qui revient à ajouter une nouvelle classe avec l
interface.

Voir exercice.

Constructeurs de classes

En Ocaml, on ne peut créer un objet d'une classe que par
`new`. Pour obtenir plusieurs constructeurs de la même clas

- utiliser des fonctions auxiliaires de construction.

  Celles-ci ne peuvent pas

- utiliser des classes auxiliaires.

  Ells peuvent être héritées, mai

- utiliser des wrappers ?

  Ils ne peuvent pas partager les variabl

- mettre les constructeurs dans des méthodes privées et u
  clauses `initializers` pour les sélectionner.

- utiliser un seul constructeur en utilisant des ar
  optionnels nommés ou étiquettés.

Utilisation de fonctions auxiliaire

On définit une classe générale

```
class c x0 y0 =
  object (* class principale *)
    val x = x0
    val y = y0
    method peu_importe = if x then y else 0
  end;;
```

puis des fonctions de création d'objets de cette cla

```
let c0 () = new c false 0
and c1 b = new c b 0
and c2 b x = new c (b && x > 0) (x * x);;
let p = c2 true 1 and q = c0();;
```

Ces constructeurs ne peuvent pas être hé

Utilisation de classes auxilliaires

On définit des variantes de la classe générale

```
class c0 = c false 0
class c1 b = c b 0
class c2 b z = c (b && z > 0) (z * z);;
let p = new c2 true 1 and q = new c0;;
```

Les classes utilisées comme constructeurs peuvent être hér

```
class c2' b x = object
  inherit c2 b x
  initializer Printf.printf "générale3 val y =
end;;
let _ = new c2' true 3;;
```

Ces constructeurs peuvent être hérités, mais individuellement, sans partage du code ajouté.

Utilisation d'un wrapper

Contrainte les fonctionnalités ajoutées n'accèdent pas au
d'instances...

Une classe et ses constructeurs

```
class wc = object
  method une_autre = Printf.printf "indépendant
class c0' = object inherit c inherit une_autre
class c1' x = object inherit c x inherit une_au
```

Situation idéale... mais assez rare

Utilisation de méthodes privées

On met les constructeurs dans la classe sous forme de mét
qui seront hérités. On utiliser les initializer pour appeler t
constructeur. Cela fonctionne bien pour des champs mutab
des valeurs par défaut.

```
class c = object
  val mutable x = 0
  method private c0 = x <- 10
  method private c1 x0 = x <- x0
end;;
class c0 = object (s) inherit c initializer s#c
class c1 z = object (s) inherit c initializer s
```

On peut hériter des constructeurs,

mais pas changer leur type.

Méthodes privées avec héritage

```
class c' = object
  inherit c as s
  val mutable y = 0
  method private c0 = s#c0; y <- 100
  method private c1' x0 y0 = s#c1 x0; y <- y0
end;;
class c0 = object (s) inherit c' initializer s#
class c1 x y =
 object (s) inherit c' initializer s#c1' x y en
```

Constructeur unique

Le principe : On distingue les différents cas par un type

```
type arg_c = Zéro | Un of bool | Deux of bool *
class c args =
  let x0, y0 = match args with
    | Zéro -> false, 0
    | Un b -> b, 0
    | Deux (b,x) -> b, x in
  object
    val x = x0 val y = y0
    method peu_importe = if x then y else 0
  end ;;
```

Problème c'est très lourd...

Solution Arguments (nommés) optionnels ou étiquettés.

Class avec arguments étiquetés

Voir la construction correspondante de Ocaml.

Ils évitent la déclaration de types concrets éphémères :

```
class point arg =
  let x0 =
    match arg with `Int x -> float x | `Float x
  object
    val x = x0
    method getx = x
  end ;;
let p = new point (`Int 2);;
let q = new point (`Float 2.5);;
```

Classe avec arguments optionnel

Voir la construction correspondante de Ocaml.

```
class c ?b:(x0 = false) ?x:(y0 = 0) =
  object
    val x = x0 val y = y0
    method peu_importe = if x then y else 0
  end ;;
```

Héritage avec arguments optionnels

```
class c' ?b ?x = object
  inherit c ?b ?x
  initializer Printf.printf "new c' with val y =
end ;;
```

```
let p = new c and q = new c true 3;;
```

Abstraction et héritage

La combinaison des modules et des classes devient inconto
y a un besoin simultané d'abstraction et d'héritage :

- Les modules réalisent l'abstration par excellence.
- Les classes permettent l'héritage avec son mécanisme d
  tardive.

Fonctions et classes amies

Le fait de cacher les variables d'instances dans les objets f
mécanisme d'abstraction de valeur. On peut ajuster cette
rendant visible en écriture ou en lecture certaines parties
méthodes appropriées.

Toutefois, l'abstraction offerte par les objets est du tout o

- Tous les objets ont accès à la partie rendue visible.
- Aucun autre objet que self n'a accès à la partie restant
  Pas même les objets de la même classe.

Une solution : avoir des amis de confiance :

- rendre la représentation visible pour comminiquer entre
- rendre la représentation abstraite (les amis ont la mêm
  sécurité, en utilisation les modules.

Les objets amis d'une même monn

```
module type MONNAIE = sig
  type t
  class m : float -> object ('a)
    method plus : 'a -> 'a
    method prod : float -> 'a method v : t
  end
end
module Monnaie = struct
  type t = float
  class m x = object (_ : 'a)
    val v = x method v = v
    method plus(z:'a) = {< v = v +. z#v >}
    method prod x = {< v = x *. v >}
  end
end;;
```

Objets amis via l'abstraction des mo

La solution repose entièrement sur les modules et fonction
la version modulaire de la monnaie

```
module Euro = (Monnaie : MONNAIE);;
let cent = new Euro.m 100.0;;
let deux_cents = cent # prod 2.0;;

module Dollar = (Monnaie : MONNAIE);;
let box = new Dollar.m 1.0;;

box # plus deux_cents;;
```

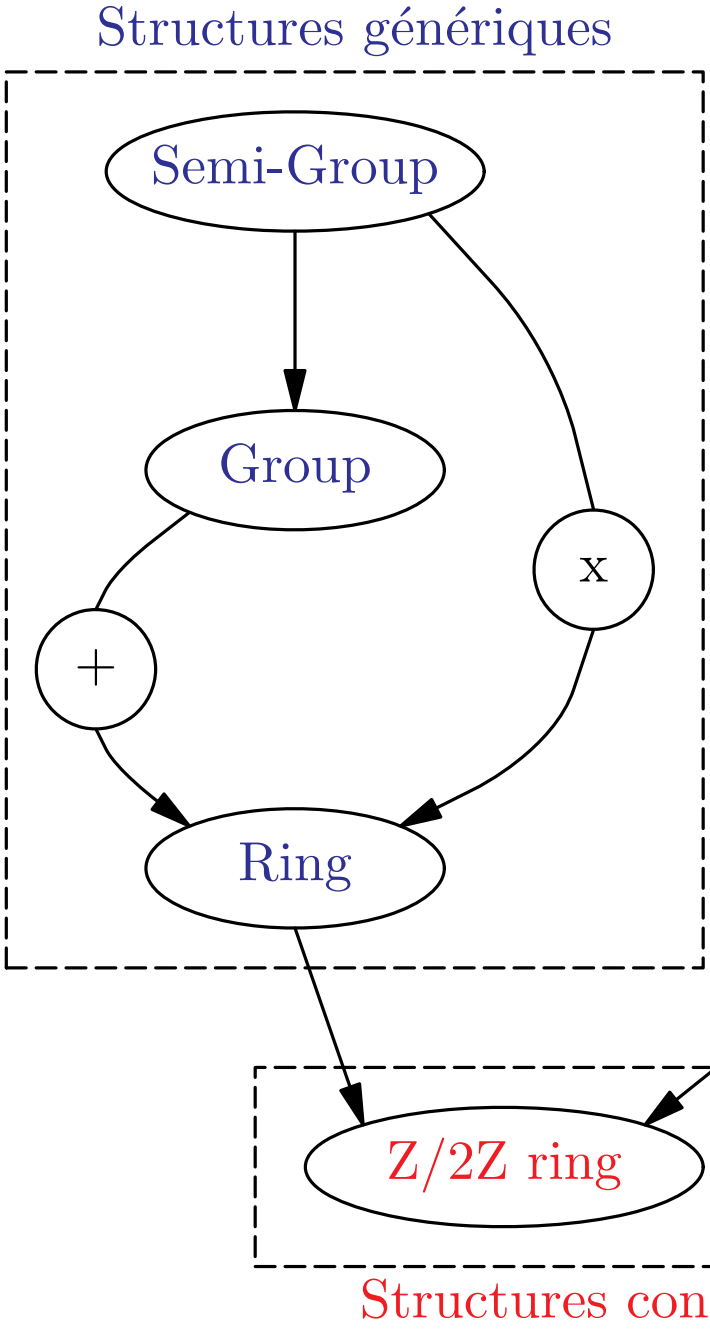
Un autre exemple :

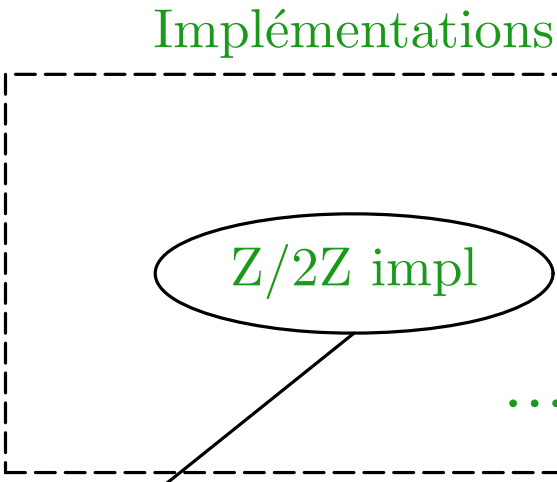
- Les ensembles avec une opération d'union.
- Les tables de hachage avec une opération de fusion.

Un exemple de combinaison in

Structures Alg

(Projet FOCS, P





- Héritage m
- Liaison tar
- Pas d'encap
- Les module

Solutions envisagées

À base de classes

- Difficulté de l'abstraction de la représentation.
- Dissymétrie des méthodes binaires

À base de modules

- Absence d'héritage (définition incrémentale)
- Absence de liaison tardive (ré-implémentation plus effic
  implémentation par défaut).

Solution retenue Mixte

- Les classes fournissent l'héritage et la liaison tardive.
- Les modules fournissent l'abstraction (et aide à l'organi

Solution combinée

Les structures algébriques peuvent être implémentées de fa
incrémentales par de petits enrichissements.

Pour pouvoir étendre et spécialiser une implémentation pa
présente l'ensemble des opérations dans une classe sans va
d'instance (essentielles).

Chaque espèce algébrique peut être présentée dans une str

1. Les opérations par défaut fournies dans une classe virt
2. Le type des opérations manquantes (type de class)
3. Le type d'une instance de cette espèces après que tout
   opérations manquantes auront été fournies.
4. Un wrapper (sous-forme de foncteur) qui assemble les
   défaut et les opérations fournies et cache l'implémenta
   résultat final.

Structures Génériques

Les structures génériques spécifient les opérations de bases
des opérations dérivées.

```
class virtual ['a] semi_group =
  object(self)
    method virtual equal: 'a -> 'a -> bool
    method not_equal x y = not (self#equal x y)
    method virtual zero: 'a
    method virtual plus: 'a -> 'a -> 'a
  end;;
```

Le paramètre `'a` est le type de la représentation des éléme
structure.

Structures générique (héritage)

Les structures génériques sont construites par héritage.

```
class virtual ['a] group =
  object(self)
    inherit ['a] semi_group
    method virtual opposite: 'a -> 'a
    method minus x y = self#plus x (self#opposit
  end;;
```

Elles utilisent la liaison tardive pour définir ou redéfinir d
spécifiés ou fournies avec une implémentation par défaut.

Structures concrètes

Ce sont des instances des structures génériques qui ont de implémentations concrètes pour les opérations de base.

Le groupe des entiers modulo p son implémemtation

```
class z_pz_impl p = object
    method equal (x : int) y = (x = y)
    method zero = 0
    method plus x y = (x + y) mod p
    method opposite x = p - 1 - x
  end;;
```

est combinée à la structure abstraite :

```
class z_pz p = object
    inherit [int] group
    inherit z_pz_impl p
  end;;
```

Abstraction de la représentation

Par les modules (on ajoute une injection une projection)

```
module type GROUP = sig
  type t
  val meth: t group
  val inj: int -> t val proj : t -> int
end;;
module Z_pZ (X: sig val p : int end) : GROUP =
  struct
    type t = int
    let meth = new z_pz 2
    let inj x =
      if x >= 0 && x < X.p then x else failwith "
    let proj x = x
  end;;
```

Motif de programmation

On peut améliorer l'exemple en définissant un motif de pr
i.e. en élaborant un modèle d'assemblage des structures g
concrètes que reproduite pour chaque type de struture.

On peut aussi augmenter la réutilisabilité en fournissant f
opérations sous forme de méthodes privées, ce qui permet
elles sont alors rendues publiques juste avant la création d
représentant les structures concrètes.