MdP : COURS4

Modèles de Programmation

Emmanuel Chailloux

PLAN

- Modules
  - modules simples
  - modules paramétrés
- Compilation séparée
- Ouverture et inclusion d'un module

Programmation modulaire

- découpage en *unités logiques* plus petites;

But: réalisation d'un module séparément des autres modules
Mise en œuvre: un module possède une *interface*, la vérification des interface est effectuée à l'assemblage des différents modules.
Intérêts:

- découpage logique;
- abstraction des données (spécification et réalisation);
- indépendance de l'implantation;
- réutilisation.

Compilation séparée

- découpage en *unités de compilation*, compilables séparément

programmation modulaire $\neq$ compilation séparée
les 2 approches sont nécessaires:

- Pour cela la spécification d'un module doit être vérifiable par un compilateur :
  - on se limite à la vérification de types
  - l'interface sera spécification de modules
  - et contiendra l'information de typage et de compilation pour les autres modules

Langage de modules d'OCaml

2 parties:

- *structure* : pour la partie réalisation/implantation
- *signature* : pour la partie spécification/interface

Le langage de modules est indépendant du langage de base.

Parallèle entre: le langage de base (*valeur : type*)
et le langage de module (*structure : signature*)!!!

Modules simples

Implantation: d'un module est une suite de définitions

- de valeurs y compris fonctionnelles
- de types
- d'exceptions
- de sous-modules

Spécification: d'un module est une suite de déclarations et de spécifications de types.

Notation: une signature sera écrite en MAJUSCULE et une structure en Minuscule dont l'initiale est en majuscule.

Implantation d'un module Queue

```
1  module Queue =
2
3  struct
4
5   type 'a t = 'a list ref
6
7   let create() = ref []
8
9   let enq x q = q:= !q@[x]
10
11  let deq q =
12   match !q with
13       [] -> failwith"Empty"
14     | h::r -> q:=r; h
15
16  let length q = List.length !q
17
18 end ;;
```

Synthèse d'une signature

L'exemple précédent donne la signature suivante :

```
1  module Queue :
2  sig
3    type 'a t = 'a list ref
4    val create : unit -> 'a list ref
5    val enq : 'a -> 'a list ref -> unit
6    val deq : 'a list ref -> 'a
7    val length : 'a list ref -> int
8  end
```

Modules: déclarations encapsulées

modules simples (structures) ⇓ ensemble de définitions

leurs types (signatures) ⇓ ensemble de spécifications de types

```
1  module Example =                    (* signature inferee *)
2  struct                               sig
3    type t = int                         type t = int
4    module M =                           module M :
5     struct                                sig
6      let succ x = x+1                      val succ : int -> int
7     end                                   end
8    let two = M.succ(1)                  val two : int
9  end                                  end
```

Accès aux éléments d'un module (1)

L'accès à un élément d'un module se fait par la notation "point".

```
# Queue.enq;;
- : 'a -> 'a list ref -> unit = <fun>
```

Y compris pour les champs d'enregistrements :

```
#  module Toto =   struct type t = {x:int; y:int}  end;;
module Toto : sig type t = { x: int; y: int } end
# let u = {Toto.x=3; Toto.y=18};;
val u : Toto.t = {Toto.x=3; Toto.y=18}
```

Ce qui peut être simplifié par l'**ouverture** du module :

```
# open Queue;
# let q = Queue.create() in ( enq "Bob" q; q);;
- : string list ref = {contents = ["Bob"]}
```

Accès aux éléments d'un module (2)

```
1  # Example.two;;
2  - : int = 2
3
4  # Example.M.succ;;
5  - : int -> int = <fun>
6
7  # Example.M.succ (Example.two);;
8  - : int = 3
```

Déclaration d'une signature

```
1  module type QUEUE =
2    sig
3      type 'a t = 'a list ref
4      val create : unit -> 'a list ref
5      val enq : 'a -> 'a list ref -> unit
6      val deq : 'a list ref -> 'a
7      val length : 'a list ref -> int
8    end
```

Quand une signature est associée à une structure il y a vérification de la cohérence :

- les déclarations de la signature existent dans la structure
- et satisfont les spécifications de la signature.

```
1  Module Queue : QUEUE = struct ...    end;;
```

signatures

La signature `ABS` n'exporte pas :

- la représentation du tye `t`,
- le module interne `M`.

```
1  module Example =                    module type ABS =
2  struct                              sig
3    type t = int                        type t
4    module M =
5       struct ... end
6    let two = M.succ(1)                  val two : t
7  end                                 end
```

Rectriction par une signature

Le nouveau module `Abs` est une *vue restreinte* de `Example`: il montre les composants de l'interface `ABS`.

```
1  # module Abs = (Example : ABS);;
2
3  # Abs.two;;      (* t devient abstrait  *)
4  - : Abs.t = <abstr>
5
6  # Abs.M.succ;;  (* M est caché  *)
7  Unbound value Abs.M.succ
```

Syntaxe du langage de modules (1)

- valeur : `module Nom [ : SIGNATURE ] =`

```
1          struct
2              val ...
3              type ...
4              exception ...
5              module ...
6          end
```

- type : `module type Nom =`

```
sig
  ...
end
```

Syntaxe du langage de modules (2)

- abstraction (valeur fonctionnelle)

```
1    module Nom =
2    functor ( Module : SIGNATURE) ->
3      struct ...
4      end
```

- application

```
1      module Nom = Module(Structure)
```

Syntaxe du langage de modules (3)

- déclaration de modules récursifs

```
1  module rec Nom [ : SIGNATURE ] =
2
3    struct ... end
4
5  and   Nom [ : SIGNATURE ] =
6
7    struct ... end
8
9  ...
```

Attention, limitation sur les dépendances croisées de calcul.

Communication entre modules

Utilisation: de déclarations d'autres modules

celle-ci peut être effectuée de 2 manières.

- **communication implicite:** en utilisant la notation "point" et en tenant compte de l'environnement global
- **communication explicite:** en utilisant des foncteurs (modules paramétrés par d'autres modules).

Communication implicite

```
1  module Element =  struct  type t = int  end;;
```

```
1  module QueueV2 =
2    struct
3      type element = Element.t
4
5      type queue = element list ref
6
7      exception Empty
8
9      let create() = ((ref []) : queue)
10
11     let enq x (q:queue) = q:= !q@[x]
12
13     let deq (q:queue)  =
14       match !q with
15         [] -> raise Empty
16       | h::r  -> q:=r; h
17   end;;
```

Signature

```
# module type QUEUEV2 =
  sig
    type element = Element.t
    and queue = element list ref
    exception Empty
    val create : unit -> queue
    val enq : element -> queue -> unit
    val deq : queue -> element
  end;;

# module QueueV3 = (QueueV2 : QUEUEV2);;
module QueueV3 : QUEUEV2
# let q = QueueV3.create() in (QueueV3.enq 18 q; q);;
- : QueueV3.queue = {contents = [18]}
```

Paramétrisation et liens

modules paramétrés (foncteurs)
⇓
fonctions sur les modules

liens des modules
⇓
foncteurs appliqués

```
1  module FunEx =                                          FunEx(↩
       Example)
2    functor (X : ABS) ->
3      struct val p = X.two .. end
```

Communication explicite

```
1  module type ELEMENT =    sig   type t   end;;
```

```
1  module QueueFunc = functor (Element : ELEMENT) ->
2      struct
3      type element = Element.t
4
5      type queue = element list ref
6
7      exception Empty
8
9      let create() = ((ref []) : queue)
10
11     let enq x (q:queue) = q:= !q@[x]
12
13     let deq (q:queue)  =
14       match !q with
15          [] -> raise Empty
16       | h::r  -> q:=r; h
17   end;;
```

Application d'un foncteur

```
# module QueueV4 = QueueFunc(Element);;
module QueueV4 : sig ... end
#  let q = QueueV4.create() in ( QueueV4.enq 44 q; q);;
- : QueueV4.queue = {contents = [44]}
```

```
# module NouvelElement =
  struct   type t = float end;;
module NouvelElement : sig type t = float end
# module QueueV5 = QueueFunc(NouvelElement);;
module QueueV5 :  sig ... end
#  let q = QueueV5.create() in ( QueueV5.enq 12.2 q; q);;
- : QueueV5.queue = {contents = [12.2]}
```

Abstraction de types

Déclarations de types:

- concrètes (définition de type visible)
- abstraites (représentation du type masquée)

Intérêts de l'abstraction de types:

- indépendance de l'implantation
- limitation du polymorphisme

Exemple d'abstraction de types

2 Définitions pour les chaînes:

```
1  module type CHAINEGEN =
2    sig  type t  val create : string -> t  end;;
3  module Chaine : CHAINEGEN = struct
4    type t = string
5    let  create (s:string) = ((String.copy s):t) end;;
6  module Maj : CHAINEGEN = struct
7    type t = string
8    let  create (s:string) = ((String.uppercase s):t) end;;
```

```
1  #  let c1 = Chaine.create "salut";;
2  val c1 : Chaine.t = <abstr>
3  #  let m1 = Maj.create("salut");;
4  val m1 : Maj.t = <abstr>
```

Suite de l'exemple

```
# c1 = m1;;
This expression has type Maj.t
 but is here used with type Chaine.t
```

```
# module QueueChaine  = QueueFunc(Chaine);;
# module QueueMaj = QueueFunc(Maj);;
```

```
# let q = QueueChaine.create() in
    ( QueueChaine.enq (Chaine.create "Alicia") q; q);;
- : QueueChaine.queue = {contents = [<abstr>]}
#  QueueChaine.enq (Maj.create "Bob") q;;
This expression has type Maj.t but is here used with type
  QueueChaine.element = Chaine.t
```

Plus d'abstraction

On définit une signature qui abstrait le type `element` :

```
1  module type NQUEUE = sig
2    type element
3    type queue
4    exception Empty
5    val create : unit -> queue
6    val enq : element -> queue -> unit
7    val deq : queue -> element
8  end
```

Perte du partage

```
1  # module QueueFuncAbs =
2    ( QueueFunc :
3          functor (E : ELEMENT) -> (NQUEUE));;
4  module QueueFuncAbs : functor (E : ELEMENT) -> NQUEUE
5  # module QueueV8 = QueueFuncAbs(Element);;
6  ...
7  # let q = QueueV8.create();;
8  val q : QueueV8.queue = <abstr>
9  # QueueV8.enq 2 q;;
10 This expression has type int but is here used with type
11   QueueV8.element = QueueFuncAbs(Element).element
```

Le compilateur a une représentation abstraite du type `element`.

Types manifestes

Contraintes de partage: entre 2 types abstraits provenant de 2 modules
C'est le cas entre `QueueV8.element` et `Element.t`.

Pour cela on indiquera une contrainte de partage dans une signature :

signature **with type** $\tau_1 = \tau_2$

```
1  module QueueFuncAbs =
2    ( QueueFunc :
3          functor (E : ELEMENT) ->
4                  (NQUEUE with type element = E.t));;
```

Exemple

```
# module QueueV9 = QueueFuncAbs(Chaine);;
module QueueV9 : ...
# let bob = Chaine.create "Bob";;
val bob : Chaine.t = <abstr>
# let q = QueueV9.create ();;
val q : QueueV9.queue = <abstr>
# QueueV9.enq bob q;;
- : unit = ()
```

Compilation séparée (1)

Unité de compilation: 2 *fichiers*

- 1 fichier d'interface (`.mli`) + 1 fichier d'implantation (`.ml`)

Sans précision:

```
1  module Nom = (
2      struct
3        contenu du fichier nom.ml
4      end :
5      sig
6        contenu du fichier nom.mli
7      end)
```

Compilation séparée (2)

correspondance: nom de module et nom de fichier

- module `Nom` correspond aux fichiers `nom.ml` et `nom.mli`
- environnement de typage : répertoires d'accès aux fichiers

Compilation séparée (3)

fichier interface: : `queue.mli`

```
1  type 'a t
2  exception Empty
3  val create : unit -> 'a t
4  val add : 'a -> 'a t -> unit
5  val push : 'a -> 'a t -> unit
6  val take : 'a t -> 'a
7  val pop : 'a t -> 'a
8  val peek : 'a t -> 'a
9  val top : 'a t -> 'a
10 val clear : 'a t -> unit
11 val copy : 'a t -> 'a t
12 val is_empty : 'a t -> bool
13 val length : 'a t -> int
14 val iter : ('a -> unit) -> 'a t -> unit
15 val fold : ('b -> 'a -> 'b) -> 'b -> 'a t -> 'b
16 val transfer : 'a t -> 'a t -> unit
```

Compilation séparée (4)

fichier implantation: : `queue.ml`

```
1  exception Empty
2
3  type 'a cell = { content: 'a; mutable next: 'a cell }
4  type 'a t = { mutable length: int; mutable tail: 'a cell }
5  let create () = { length = 0; tail = Obj.magic None }
6  let clear q = q.length <- 0; q.tail <- Obj.magic None
7  let add x q =
8    q.length <- q.length + 1;
9    if q.length = 1 then
10     let rec cell = { content = x; next = cell } in
11     q.tail <- cell
12   else
13     let tail = q.tail in
14     let head = tail.next in
15     let cell = { content = x; next = head } in
16     tail.next <- cell; q.tail <- cell
17
18 let push = add
19
20 let peek q =
21   if q.length = 0 then
22     raise Empty
```

Compilation séparée (5)

Compilation:

```
% ocamlc -c queue.mli
% ocamlc -c queue.ml
```

Fichiers objet:

```
% ls queue.cm?
queue.cmi       queue.cmo
```

Compilation séparée (6)

utilisation:

```
1  let q = Queue.create();;
2  let r = Queue.create();;
3
4  let main() =
5    Queue.add 3 q ; Queue.add 4 q ;
6    Queue.add "Ping" r; Queue.add "Pong" r;
7    print_int (Queue.take q); print_int (Queue.take q); ↩
         print_newline();
8    print_string (Queue.take r); print_string (Queue.take r)↩
         ; print_newline();;
9
10 main();;
```

compilation:

```
% ocamlc queue.cmo main.ml -o main.exe
```

Exécution:

```
% ./main.exe
34
PingPong
```

Ouverture d'un module

- global
  Syntaxe: **open** mod-name;;
  Racourci: de la notation "point"
  Exemple:

```
1 # open QueueV9;;
2 # let q = create();;
3 val q : QueueV9.queue = <abstr>
```

- local
  Syntaxe: **let open** mod-name **in** expr

Inclusion d'un module

Syntaxe: **include** mod-expr;;

réexportation dans la structure courante des définitions de `mod-expr`

Exemple:

```
1  module QueueV2B = struct
2    include QueueV2
3    let length (q : queue) = List.length !q
4  end;;
5
6  module QueueV2B :
7    sig
8      type element = Element.t
9      type queue = element list ref
10     exception Empty
11     val create : unit -> queue
12     val enq : element -> queue -> unit
13     val deq : queue -> element
14     val length : queue -> int
15   end
```

Différence entre open et include

- open crée des raccourcis des chemins des définitions d'une structure sans rien définir localement;
- include ajoute les définitions du module inclus dans les définitions du module courant

```
1  module QueueV2C = struct
2    open QueueV2
3    let create = create
4    let enq = enq
5    let deq = deq
6    let length (q : queue) = List.length !q
7  end;;
8
9  module QueueV2C :
10   sig
11     val create : unit -> QueueV2.queue
12     val enq : QueueV2.element -> QueueV2.queue -> unit
13     val deq : QueueV2.queue -> QueueV2.element
14     val length : QueueV2.queue -> int
15   end
```

Sous-modules

Définitions: de modules dans un module

Intérêts: organisation hiérarchique, visibilité des champs des modules extérieurs, nécessaire avec la compilation séparée

Exemple:

```
1  module M1 =
2    struct  type t1
3      let f = ...
4      module type SMT1 = sig type t2 = (t1,t1) ... end
5      module SM2 : SMT1 = struct let g x = f(f x) ... end
6    end;;
```

Accès:

```
1  M1.SM2.g;;
```

Modules locaux

Syntaxe: **let module** mod-name = mod-expr in expr

Intérêt: création dynamique (à l'exécution) de modules

Exemple: appliquer un foncteur sur une *structure* dont l'un des champs est un paramètre d'une fonction.

```
# let g  (l : string list) =
  let module Toto =
    Set.Make(struct type t = string
      let  compare a b = if a.[0] < b.[0] then -1
        else if a.[0] > b.[0] then 1 else 0  end)
  in Toto.min_elt
    (List.fold_right Toto.add l Toto.empty);;
val g : string list -> string = <fun>
```

Foncteurs de foncteurs

Plusieurs paramètres: à un module paramétré

Intérêts: paramétrage d'un module application par plusieurs modules, création de squelettes, y compris avec abstraction de types et contraintes de partage.

Exemple: jeu à 2 joueurs

```
1 module Jeu =
2   functor (Rep : REPRESENTATION) ->
3   functor (Aff : AFFICHAGE) ->
4   functor Alphabeta : ALPHABETA ->   struct ... end;;
5 module Main =  Jeu (Stone_rep) (Stone_graph (Stone_rep))
6                    (Alphabeta (Stone_rep)) ;;
7 Main.main() ;;
```