Modules

Modules et interfaces

- rôle des interfaces: *masquer*
  - cacher des valeurs: une valeur existe, mais on ne peut en parler (pas de nom)
  - *type abstrait*: déclarer l'existence d'un type en en masquant l'implémentation (seul un nom est fourni)
- utilisation dans le développement logiciel
  - deux modules, une interface

    implémentations différentes d'une même structure de données

    DÉMO `complexes.ml`
  - deux interfaces, un module (sans type abstrait)

    DÉMO `gensym-mod.ml`

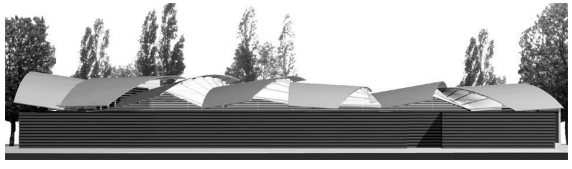
    ↪ aspect "sécurité" – cf. aussi l'encapsulation d'un état (accès contrôlé aux effets de bord)
  - différentes vision d'un même module

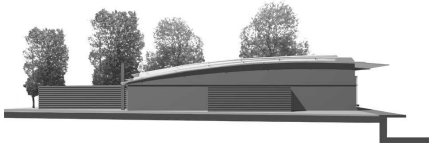
Un module avec un type abstrait

```
module M =
  ( struct
    type t = int ref
    let create() = ref 0
    let add x = x:=!x+1
    let get x = if !x>0
       then (x:=!x-1; 1)
       else failwith "Empty"
  end
```

```
sig
  type t
  val create :  unit -> t
  val add :  t -> unit
  val get :  t -> int
end )
```

projeter suivant des *vues*





M. Mimram, *Piscine à Viry-Chatillon*

Deux vues sur le module `M`

```
 module type S1 =     (* propriétaire *)
  sig
    type t
    val create :  unit -> t
    val add :  t -> unit
  end

module type S2 =     (* utilisateur *)
  sig
    type t
    val get :  t -> int
  end

module M1 = (M:S1);;
module M2 = (M:S2);;
let x = M1.create() in M1.add x ; M2.get x ;;

  This expression has type M1.t but is here used with type M2.t
```

masquer + “déraciner”

Modifier une signature à la volée

- la construction `with`

```
module M1 = (M:S1 with type t = M.t) ;;
module M2 = (M:S2 with type t = M.t) ;;
let x = M1.create() in M1.add x ; M2.get x ;;
  - :  int = 1
```

- d'ailleurs

```
module type Order =
   sig
     type t
     val order :  t->t->bool
   end
```

```
module IntOrder :  Order =
   struct
     let minus x y = x-y
     let order x y =
         (minus y x) >= 0
     type t = int
   end
```

  - `IntOrder.order 3 4` $\rightsquigarrow$ erreur `IntOrder` est inutilisable
  - `module IntOrder2 :  (Order with type t = int) ...`

Cohérence entre les types

- ça non plus, ça ne marche pas:

```
module M1 = (M:S1) ;;
module M2 = (M:S2 with type t = M1.t) ;;
  Type declarations do not match:
    type t = M.t is not included in type t = M1.t
```

- on construit des types dont les *noms* diffèrent, bien qu'ils soient les mêmes *"en dessous"*
- par défaut, ces types sont incompatibles, sauf si on explicite le partage (`with`)
  - le problème surgit lors de la phase de typage
  - on a affaire à un **choix de conception** du langage

Modules et interfaces, donc

- quand on définit un module, il a une interface par défaut

  ```
  module M = struct...end
  ```

- définir un module en indiquant explicitement une signature, c'est *"sceller"* la signature sur le module

  ```
  module type S = type t val f :  int -> t end
  module M1 :  S = struct ...end
  ```
  ou bien `module M2 = (M:S)`
  - un utilisateur de `M1` ou `M2` n'a pas le droit de savoir comment `t` est implémenté

- on peut utiliser `with` pour modifier une signature au vol, et "ouvrir des fenêtres" dans le sceau qu'on a collé au module

  ```
  module M : S with type t = string = struct ...end
  ```

Foncteurs

Modules paramétrés: les foncteurs

- `module Machin (M:Truc) = struct ...end`
  - des 'fonctions sur les modules'
  - avec typage explicite du module passé en argument
- DÉMO `fonct.ml`
- un exemple important de foncteur: la librairie `Set`

  DÉMO `demo_set.ml`

Autre exemple de foncteur: les listes triées

```
    (* liste triée *)
module type OrderedList =
   sig
     module O:Order
     type t
     val nil:  t
     val cons:  O.t -> t -> t
     exception Empty
     val hd:  t -> O.t
     val tl:  t -> t
     val merge:  t -> t -> t
     val from_list :  O.t list -> t
   end
```

Les foncteurs (2)

définitions globales:

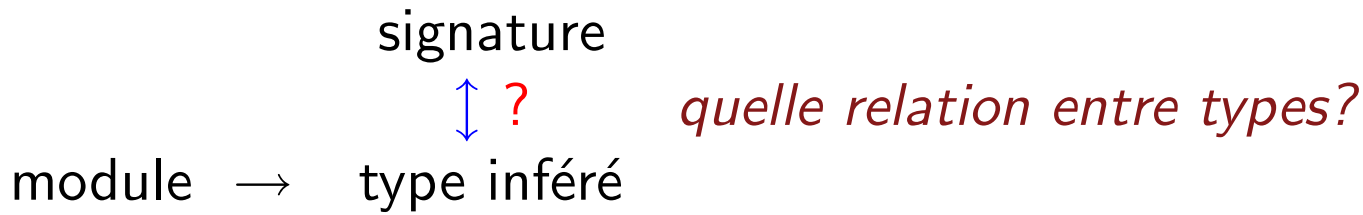
```
type bin_tree = Leaf | Node of ...
heap_insert...heap_merge...heap2list...
```

```
module ListByHeap (X:Order) :  (OrderedList with module O=X) =
   struct
     module O=X
     type t = O.t bin_tree
     let nil = Leaf
     let cons x l = heap_insert O.order x l
     let merge l m = heap_merge O.order l m
     let from_list l = heap2list O.order l
     exception Empty (* REDEFINIR Empty ICI *)
     let hd l = match l with | Node (x,_,_) -> x
                             | Leaf -> raise Empty
     let tl l = match l with | Node (_,a,b) -> heap_merge a b
                             | Leaf -> raise Empty
   end
```

Sous-typage

Modules et types

- un module peut être vu comme un agrégat de code
- signature: le type associé à cet agrégat (“type du module”)
- le scénario:



- on a déjà vu les *équivalences entre types*
  - `(t1*t2)->t3` $\leftrightarrows$ `t1->t2->t3`
  - `t*t` $\not\leftrightarrows$ `t`
  - “la même information” dans les deux types

La relation de sous-typage

- si `t` et `t'` sont deux types, on écrit

  $t \leq t'$ *un* `t` *peut être vu comme un* `t'`

  pour exprimer le fait que `t` est *moins général* que `t'`
  - avec les modules `t`: type du module, `t'`: type de la signature
  - si pas de signature (ou de `.mli`): $\leq$ c'est $=$
- $\leq$: à quoi a-t-on droit?
  - donner le "vrai" type d'une valeur, ou un "vrai" type

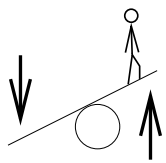
    `val f : int -> bool` `type coord = int*int`
  - "oublier" une valeur
  - masquer un type `type t` *(un peu comme un $\exists$)*
  - "battre les cartes" (les champs dans le mauvais ordre)
  - perdre de la généralité dans le polymorphisme (`'a` $\rightsquigarrow$ `int`)

Sous-typage, exemples

- `(sig type t=int end)`
  $\not\leq$ `(sig type t=int  val v:t end)`
- `(sig type t=int  val v:t end)`
  $\leq$ `(sig type t=int end)` $\leq$`(sig end)` + gde sign.
- `(sig type t=int  val v:t end)`
  $\leq$ `(sig type t val v:t end)`
- `(sig type t=int*bool end)`
  $\not\leq$ `(sig type t=int end)`
- `(sig type t=int  val v:t end)`
  $\leq$ `(sig val v:int end)`
- `(sig type t  type u=t end)`
  $\leq$ `(sig type u  type t=u end)`
- plus petite signature?
  `sig val x : int val x : bool end`
  ou alors `sig` ⟨*une infinité de valeurs*⟩ `end` "le faux" (`sig end` étant "le vrai")

Foncteurs et sous-typage

- $t \leq t'$: `t` est *moins général* que `t'`, `t` *peut être vu comme* `t'`
- à quelle condition
  `functor (X:U1) -> T1` $\leq$ `functor (X:U2) -> T2` ?
  - on a `functor (X:U)->T1` $\leq$ `functor (X:U)->T2`
    si $\boxed{\texttt{T1} \leq \texttt{T2}}$
    p.ex. `T1 = struct val a:int val b:bool end`
    `T2 = struct val b:bool end`
  - `functor (X:U1)->T` $\leq$ `functor (X:U2)->T`
    si $\boxed{\texttt{U2} \leq \texttt{U1}}$
    p.ex. `U1 = struct val b:bool end`
    `U2 = struct val a:int val b:bool end`
- `->` est *covariante à droite, contravariante à gauche*



($A \Rightarrow \top$, $\bot \Rightarrow A$ en logique)

Sur le typage: spécification et implémentation

- la relation entre interfaces et modules
  - ▷ *spécification*: les services que le module doit fournir
  - ▷ *implémentation*: les services fournis par un module donné
  - ▷ "cahier des charges tenu": impl. satisfait spécif.
- spécification/implémentation: plus généralement, vérifier une propriété d'un programme donné
- des types *"sanctions"* aux types *"point de vue"*

analyse statique: nombreux travaux de recherche à l'heure actuelle *(types pour: code mort, sécurité, complexité, . . .)*

Aspects de la modularité en C

- en OCaml, les modules sont *dans le langage*
  modules, interfaces, valeurs, types, types abstraits
- en C: fichiers `.c` et `.h` *(header)*
  - dans `toto.h`: `int f(int x, int y)`
  - dans `toto.c`: `int f(){...}`
  - dans `prog.c`: `#include "toto.h" ...f(32,52)...`

  - `gcc -c toto.c` $\rightsquigarrow$ `toto.o` *(pas de vérification)*
  - `gcc -c prog.c` $\rightsquigarrow$ `prog.o`
  - `gcc toto.o prog.o -o prog`
    $\rightsquigarrow$ **n'importe quoi à l'exécution**
- pratique courante: inclure le `.h` dans le `.c` (`#include "toto.h"`), pour forcer à typer
  - "le `.cmi` du pauvre"

- on peut faire `type toto` *(tout court)* dans un `.h`, mais on ne peut manipuler que des `toto*` à l'extérieur

  DÉMO `t_abstr_main.c, type_abstrait.(h,c)`

Pour finir

- les points dont on a discuté s'apparentent à *des choix de conception* lorsque l'on fabrique un langage de programmation:
  - les questions de modularité doivent-elles être réglées...
    - plus ou moins empiriquement (à la compilation, éventuellement en s'appuyant sur des directives ayant trait à l'interfaçage et à la liaison), ou bien...
    - au sein du langage (comme dans OCaml, mais aussi comme dans les langages orientés objet *(cf. héritage multiple)*, ou la programmation par composants)?
  - comment règle-t-on les questions de cohérence?
  - ...
- ces questions (cacher l'information, gérer la cohérence...) ne sont pas propres à tel ou tel langage, elles se posent "partout"
- la programmation orientée objets propose également des solutions aux questions liées à la programmation modulaire