Cours de Programmation Avancée

L3 ENS Paris Saclay

Giuseppe Castagna

CNRS

Outline of the course

- Module systems

  - 1. Introduction to modularity. - 2. ML simple modules. - 3. Functors. - 4. Advanced example.

- Classes vs. Modules

  - 5. Modularity in OOP. - 6. Mixin Composition - 7. Multiple dispatch - 8. OCaml Classes - 9. Haskell's Typeclasses - 10. Generics

- Computational effects.

  - 11. Exceptions. - 12. Imperative features. - 13. Continuations

- Program transformations.

  - 14. The fuss about purity - 15. A Refresher Course on Operational Semantics - 16. Closure conversion - 17. Defunctionalization - 18. Exception passing style - 19. State passing style - 20. Continuations, generators, and coroutines - 21. Continuation passing style

- Abstract machines

  - 22. A simple stack machine - 23. The SECD machine - 24. Adding Tail Call Elimination - 25. The Krivine Machine - 26. The lazy Krivine machine - 27. Eval-apply vs. Push-enter - 28. The ZAM - 29. Stackless Machine for CPS terms

- Subtyping

  - 36. Simple Types - 37. Recursive types

- XML Programming

  - 38. XML basics - 39. Set-theoretic types - 40. Examples in Perl - 41. Covariance and contravariance - 42. XML Programming in CDuce - 43. Toolkit

- Concurrency

  - 44. Concurrency - 45. Preemptive multi-threading - 46. Mutexes, Conditional Variables, Monitors - 47. Doing without mutual exclusion - 48. Cooperative multi-threading - 49. Channeled communication - 50. Software Transactional Memory

- Subtyping

  - 36. Simple Types - 37. Recursive types

- XML Programming

  - 38. XML basics - 39. Set-theoretic types - 40. Examples in Perl - 41. Covariance and contravariance - 42. XML Programming in CDuce - 43. Toolkit

- Concurrency

  - 44. Concurrency - 45. Preemptive multi-threading - 46. Mutexes, Conditional Variables, Monitors - 47. Doing without mutual exclusion - 48. Cooperative multi-threading - 49. Channeled communication - 50. Software Transactional Memory

Le langage de référence pour le cours est OCaml, mais nous utiliserons aussi des extraits de code Haskell, Scala, Perl 6, C#, Java, Erlang, Pascal, Python, Basic, CDuce, Xslt, Go ... . L'idée étant de mettre l'accent sur les concepts de la programmation plus que sur la programmation dans un langage particulier.

La note finale de l'examen est calculée de la manière suivante.

(1/2 (Examen) + 1/3 max(Examen, Projet) + 1/6 Projet)*CB

[CB = if Projet > 6 then 1 else 0.7]

Modules

Outline

Outline

Les modules sont partout !

- Toute construction complexe :
  - bâtiment,
  - voiture,
  - avion,
  - compilateur,...

  suppose une *modularisation*.
- Les raisons technologiques sont évidentes (organisation du travail, fiabilité,...)
- ... et les retombées économiques immédiates.

Généralités sur les modules

- On peut construire, tester, analyser... un module de façon *indépendante* d'autres modules.
- Un module a une *interface* qui décrit ses modalités d'interaction.
- Éventuellement, une *spécification* qui décrit son comportement.
- Une *implémentation*.

Un changement d'implémentation devrait être *"transparent"* à l'utilisateur (à un certain niveau d'abstraction !).

Aspects spécifiques des modules en programmation

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 interfaces est effectuée à l'assemblage des différents modules.

Intérêts :

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

Compilation séparée

Découpage en *unités de compilation*, compilables séparément

Compilation séparée

Découpage en *unités de compilation*, compilables séparément

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

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

Programmation à petite et grande échelle

- A petite échelle (Wirth 1975) :

  Programme = Algorithme + Structures de Données

- A grande échelle :

  Module = Programme + Interface + (Spécification)

Un problème ancien. . .

Le problème dans le contexte de la programmation a été identifié depuis longtemps. Par exemple :

> *DeRemer, Kron. Programming in the large versus programming in the small. IEEE Trans. on Soft. Eng., 1976.*
>
> *Parnas. On the criteria to be used in decomposing systems into modules. CACM, 1972.*

Un problème toujours d'actualité...

- Bibliothèques,
- "Modules" en C.
- Packages en Ada et Java.
- Modules en Modula et ML.
- Programmation par composants
- Interfaces 'Web services'.

Un problème plus difficile : l'interopérabilité

Programmation modulaire On cherche à faire interagir des modules qui appartiennent au *même langage*.

Interopérabilité On cherche à faire interagir des modules de langages qui *diffèrent* dans :

- la représentation des données,
- le traitement des exceptions,
- l'organisation de la mémoire,
- ...

Outline

Les modules en ML

Nous allons étudier le système de modules de ML.

- De loin le système de modules le plus sophistiqué et le plus étudié.
- La conception du système de modules est assez *indépendant* du langage de programmation. Le système de modules de ML a été appliqué aussi à d'autres langages.
- Une généralisation du concept de type de données abstrait.

Théorie (des types) encore en développement : compliquée et pas stable. Les détails présentés ici sont basés sur OCAML et en particulier sur le Chapitre 14 de
`https://caml.inria.fr/pub/docs/oreilly-book/html/index.html`

Terminologie ML

- Structure (= Implémentation).
- SIGNATURE (= Interface).

$$\textit{Structure} : \textit{SIGNATURE} \quad \sim \quad \textit{Valeur} : \textit{TYPE}$$

Remarques

1. En ML on sépare *valeurs* et *types*. Or les structures contiennent des *types* et des *valeurs*, on ne peut donc pas les voir comme des valeurs.
2. Une *structure* (une *signature*) n'est pas une valeur (un type) de première classe.

Modules (simples)

Structure

Une suite de définitions de :

- valeurs (y compris fonctions)
- types
- exceptions
- sous-modules

SIGNATURE

Une suite déclarations de types, d'exceptions et de noms avec leur type/signature.

Modules (simples)

Structure

Une suite de définitions de :

- valeurs (y compris fonctions)
- types
- exceptions
- sous-modules

SIGNATURE

Une suite déclarations de types, d'exceptions et de noms avec leur type/signature.

Convention

On utilise, `Machin` pour une structure et `MACHIN` pour une signature.

Exemple : Structure d'un module Queue

```
module Queue =
struct
  type 'a queue = 'a list ref
  let create() = ref []
  let enq x q = q:= !q@[x]          (* horrible si @ n'est pas lazy *)
  let deq q =
    match !q with
          [] -> failwith "Empty"
        | h::r -> q:=r; h
  let length q = List.length !q       (* utilisation module List *)
end ;;
```

La signature associée

La système synthétise automatiquement la signature suivante.

```
module Queue :
sig
  type 'a queue = 'a list ref
  val create : unit -> 'a list ref
  val enq : 'a -> 'a list ref -> unit
  val deq : 'a list ref -> 'a
  val length : 'a list ref -> int
end
```

La signature associée

La système synthétise automatiquement la signature suivante.

```
module Queue :
sig
  type 'a queue = 'a list ref
  val create : unit -> 'a list ref
  val enq : 'a -> 'a list ref -> unit
  val deq : 'a list ref -> 'a
  val length : 'a list ref -> int
end
```

Dans la signature générée le type de données qui représente la queue est *visible* ainsi que l'ensemble des opérations définies avec les type le plus général.

Un autre exemple avec structures imbriquées

Valeur (structure)

```
module Example =
struct
  type t = int
  module M =
  struct
    let succ x = x+1
  end
  let two = M. succ(1);;
end;;
```

Type (signature)

```
module Example :
sig
  type t = int
  module M :
  sig
    val succ : int -> int
  end
  val two : int
end;;
```

Accès aux éléments d'un module par identificateur qualifié

- La notation 'point' :

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

- S'applique aussi aux 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}
```

Ouverture d'un module

- On peut ouvrir un module et donc accéder toutes les déclarations de la structure associée :

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

# Example.two;;
-: int = 2

# Example.M.succ;;
-: int -> int = <fun>

# Example.M.succ (Example.two);;
-: int = 3

(* une structure n'est pas une valeur *)
# Example.M;;
Error: Unbound constructor Example.M
```

Ouverture d'un module

- On peut ouvrir un module et donc accéder toutes les déclarations de la structure associée :

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

# Example.two;;
-: int = 2

# Example.M.succ;;
-: int -> int = <fun>

# Example.M.succ (Example.two);;
-: int = 3

(* une structure n'est pas une valeur *)
# Example.M;;
Error: Unbound constructor Example.M
```

L'ouverture d'un module peut cacher des déclarations locales.

Déclaration d'une signature

- On peut déclarer une *signature* comme suit :

```
module type QUEUE =
 sig
 type 'a queue = 'a list ref
 val create : unit -> 'a list ref
 val enq : 'a -> 'a list ref -> unit
 val deq : 'a list ref -> 'a
 val length : 'a list ref -> int
end;;
```

Déclaration d'une signature

- On peut déclarer une *signature* comme suit :

```
module type QUEUE =
 sig
 type 'a queue = 'a list ref
 val create : unit -> 'a list ref
 val enq : 'a -> 'a list ref -> unit
 val deq : 'a list ref -> 'a
 val length : 'a list ref -> int
end;;
```

- En associant une structure à une signature le système vérifie que tous les éléments de la signature existent dans la structure. Par exemple :

```
module Queue : QUEUE = struct...end;;
```

- La structure peut contenir un élément avec un type *plus général* que celui spécifié dans la signature.

- La structure peut contenir un élément avec un type *plus général* que celui spécifié dans la signature.
- Elle peut aussi contenir *d'avantage d'éléments* que ceux décrits dans la signature et dans un ordre différent.

```
module Example = struct
  type t = int
  module M = struct
   let succ x = x+1
  end
  let two = M.succ(1)
end ;;

module type ABS = sig
  type t
  val two : t
end;;
```

- La structure peut contenir un élément avec un type *plus général* que celui spécifié dans la signature.
- Elle peut aussi contenir *d'avantage d'éléments* que ceux décrits dans la signature et dans un ordre différent.

```
module Example = struct
  type t = int
  module M = struct
   let succ x = x+1
  end
  let two = M.succ(1)
end ;;

module type ABS = sig
  type t
  val two : t
end;;
```

- Il s'agit d'un exemple de *sous-typage* (sur les signatures). On discutera ce concept plus tard dans le cours.

- En associant une structure à une signature on ne peut utiliser que les éléments déclarés dans la signature. Ici un type *t* dont on ignore la représentation et une valeur *two* de ce type.
- Le module Abs est une restriction du module Example.

```
# module Abs = (Example: ABS);; (* Nous cachon M et t *)
module Abs : ABS

# Abs.two;;                    (* M est utilisable *)
- Abs.t = <abstr>              (* t est abstrait   *)

# Abs.M.succ(1)                (* M est invisible  *)
Unbound value Abs.M.succ
```

Types concrets et types abstraits

On définit une signature LISTE.

```
# module type LISTE = sig
    type 'a t
    val creer : unit -> 'a t
    val inserer : 'a -> 'a t -> 'a t
  end;;
```

Types concrets et types abstraits

On définit une signature LISTE.

```
# module type LISTE = sig
    type 'a t
    val creer : unit -> 'a t
    val inserer : 'a -> 'a t -> 'a t
  end;;
```

dont on propose deux *implémentations* :

Types concrets et types abstraits

On définit une signature LISTE.

```
# module type LISTE = sig
    type 'a t
    val creer : unit -> 'a t
    val inserer : 'a -> 'a t -> 'a t
  end;;
```

dont on propose deux *implémentations* :

```
# module Liste1 = struct
    type 'a t = 'a list
    let creer () = []
    let inserer x l = x::l
  end;;

module Liste1 : sig
  type 'a t = 'a list
  val creer : unit -> 'a list
  val inserer : 'a -> 'a list -> 'a list
end
```

Types concrets et types abstraits

On définit une signature LISTE.

```
# module type LISTE = sig
    type 'a t
    val creer : unit -> 'a t
    val inserer : 'a -> 'a t -> 'a t
  end;;
```

dont on propose deux *implémentations* :

```
# module Liste1 = struct
    type 'a t = 'a list
    let creer () = []
    let inserer x l = x::l
  end;;

module Liste1 : sig
  type 'a t = 'a list
  val creer : unit -> 'a list
  val inserer : 'a -> 'a list -> 'a list
end
```

```
# module Liste2 = struct
    type 'a t = 'a list
    let creer () = []
    let inserer x l = l (*bad*)
  end;;

module Liste2 : sig
  type 'a t = 'a list
  val creer : unit -> 'a list
  val inserer : 'a -> 'b -> 'b
end
```

La structure concrète des types est *visible* et on peut écrire :

```
# Liste1.creer();;
- : 'a list = []

# Liste2.inserer 3 (Liste1.creer ()) ;;
- : 'a list = []
```

La structure concrète des types est *visible* et on peut écrire :

```
# Liste1.creer();;
- : 'a list = []

# Liste2.inserer 3 (Liste1.creer ()) ;;
- : 'a list = []
```

Maintenant on *masque* Liste1 et Liste2 avec la signature LISTE :

```
# module Liste1 = (Liste1: LISTE);;
module Liste1 : LISTE

# module Liste2 = (Liste2: LISTE);;
module Liste2 : LISTE
```

La structure concrète des types est *visible* et on peut écrire :

```
# Liste1.creer();;
- : 'a list = []

# Liste2.inserer 3 (Liste1.creer ()) ;;
- : 'a list = []
```

Maintenant on *masque* Liste1 et Liste2 avec la signature LISTE :

```
# module Liste1 = (Liste1: LISTE);;
module Liste1 : LISTE

# module Liste2 = (Liste2: LISTE);;
module Liste2 : LISTE
```

Les types deviennent *abstraits* et donc *incomparables*.

```
# Liste1.creer();;
- : '_a Liste1.t = <abstr>

#  Liste2.inserer;;
- : 'a -> 'a Liste2.t -> 'a Liste2.t = <fun>

#  Liste2.inserer 3 (Liste1.creer ()) ;;
This expression has type 'a Liste1.t
but is here used with type int Liste2.t
```

Variables de types faibles

Une variable de type faible `'_a` est une variable qui ne peut pas être généralisée. Elle est instanciée par un type qui n'a pas encore pu être déterminé.

Elle apparait lorsque le compilateur Caml essaie de compiler une fonction ou une valeur qui est monomorphe, mais pour laquelle certains types n'ont pu être complètement inferés.
Elles disparait grâce au mécanisme d'inférence de types dès que suffisamment d'informations auront pu être rassemblées.

```
# let id x = x;;
val id : 'a -> 'a = <fun>

# let id2 = id id;;
val id2 : '_a -> '_a = <fun>

# let a = (id 1 , id "1");;
val a : int * string = (1, "1")

# let b = (id2 1 , id2 "1");;
Error: This expression has type string but an
       expression was expected of type int
```

Le problème vient de l'utilisation des références mutables :

```
# let r = ref []
val r : 'a list ref
# r := [3]; r
- : 'a list ref
# let l = List.map (function true -> 1 | false -> 2) !r
val l : int list = Segmentation fault
```

Le problème vient de l'utilisation des références mutables :

```
# let r = ref []
val r : 'a list ref
# r := [3]; r
- : 'a list ref
# let l = List.map (function true -> 1 | false -> 2) !r
val l : int list = Segmentation fault
```

Solution in ML (Wright '95) : seulement les *valeurs* peuvent être polymorphes (i.e., pas les applications ni les créations de cellules de mémoire).

Le problème vient de l'utilisation des références mutables :

```
# let r = ref []
val r : 'a list ref
# r := [3]; r
- : 'a list ref
# let l = List.map (function true -> 1 | false -> 2) !r
val l : int list = Segmentation fault
```

Solution in ML (Wright'95) : seulement les *valeurs* peuvent être polymorphes (i.e., pas les applications ni les créations de cellules de mémoire).

```
# let r = ref [] ;;
val r : '_a list ref
# r := [3]; r ;;
- : int list ref
# let l = List.map (function true -> 1 | false -> 2) !r;;
Error: This expression has type int list
       but an expression was expected of type bool list
```

Révenons à l'example Queue

Une manière plus raisonnable de *déclarer la signature* de Queue

```
module type QUEUE = sig
 type 'a queue
 exception Empty
 val create : unit -> 'a queue
 val enq : 'a -> 'a queue -> unit
 val deq : 'a queue -> 'a
 val length : 'a queue -> int
end;;
```

```
module Queue : QUEUE = struct
  type 'a queue = 'a list ref
  exception Empty
  let create() = ref []
  let enq x q = q:= !q@[x]
  let deq q =
    match !q with
          [] -> raise Empty
       | h::r -> q:=r; h
  let length q = List.length !q
end ;;
```

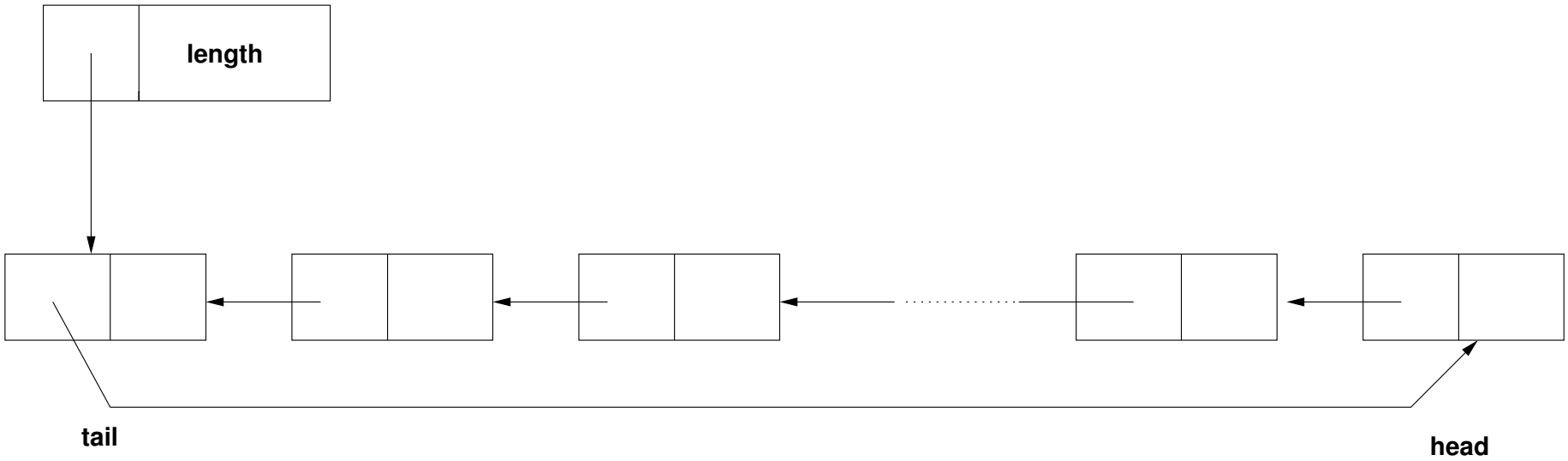
(nous avons caché que le type utilisé pour l'implémentation est `'a list ref`)

Une manière plus raisonnable de *définir* Queue

```
module Queue : QUEUE = struct
  type 'a cell = { content: 'a;  mutable next: 'a cell }    (* invisible *)
  type 'a queue = { mutable length: int; mutable tail: 'a cell }
  exception Empty
  let create () = { length = 0; tail = Obj.magic None }
  let enq x q =
    q.length <- q.length + 1;
    if q.length = 1 then
       let rec cell = { content = x; next = cell} in
       q.tail <- cell
    else
       let tail = q.tail in
       let head = tail.next in
       let cell = { content = x; next = head} in
       tail.next <- cell;
       q.tail <- cell
  let deq q =
    if q.length = 0 then raise Empty;
    q.length <- q.length - 1;
    let tail = q.tail in
    let head = tail.next in
    if head == tail then
      q.tail <- Obj.magic None
    else
      tail.next <- head.next;
    head.content
  let length q = q.length
end;;
```

Voyons cela graphiquement

Implementation de Queue



Une manière plus raisonnable de *définir* Queue

```
module Queue : QUEUE = struct
  type 'a cell = { content: 'a;  mutable next: 'a cell }    (* invisible *)
  type 'a queue = { mutable length: int; mutable tail: 'a cell }
  exception Empty
  let create () = { length = 0; tail = Obj.magic None }
  let enq x q =
    q.length <- q.length + 1;
    if q.length = 1 then
       let rec cell = { content = x; next = cell} in
       q.tail <- cell
    else
       let tail = q.tail in
       let head = tail.next in
       let cell = { content = x; next = head} in
       tail.next <- cell;
       q.tail <- cell
  let deq q =
    if q.length = 0 then raise Empty;
    q.length <- q.length - 1;
    let tail = q.tail in
    let head = tail.next in
    if head == tail then
      q.tail <- Obj.magic None
    else
      tail.next <- head.next;
    head.content
  let length q = q.length
end;;
```

Here we bypass the type-checker

```
module Queue : QUEUE = struct
  type 'a cell = { content: 'a;  mutable next: 'a cell }
  type 'a queue = { mutable length: int; mutable tail: 'a cell }
  exception Empty
  let create () = { length = 0; tail = Obj.magic None }
  let enq x q =
    q.length <- q.length + 1;
    if q.length = 1 then
       let rec cell = { content = x; next = cell} in
       q.tail <- cell
    else
       let tail = q.tail in
       let head = tail.next in
       let cell = { content = x; next = head} in
       tail.next <- cell;
       q.tail <- cell
  let deq q =
    if q.length = 0 then raise Empty;
    q.length <- q.length - 1;
    let tail = q.tail in
    let head = tail.next in
    if head == tail then
      q.tail <- Obj.magic None
    else
      tail.next <- head.next;
    head.content
  let length q = q.length
end;;
```

L'utilisation de `option` est plus propre mais 5% plus lente

```
module Queue : QUEUE = struct
  type 'a cell = { content: 'a;  mutable next: 'a cell option}
  type 'a queue = { mutable length: int; mutable tail: 'a cell }
  exception Empty
  let create () = { length = 0; tail = None }
  let enq x q =
    q.length <- q.length + 1;
    if q.length = 1 then
       let rec cell = { content = x; next = cell} in
       q.tail <- Some cell
    else
       let Some tail = q.tail in      (* non exhaustive pattern matching *)
       let head = tail.next in
       let cell = { content = x; next = head} in
       tail.next <- cell;
       q.tail <- Some cell
  let deq q =
    if q.length = 0 then raise Empty;
    q.length <- q.length - 1;
    let Some tail = q.tail in         (* non exhaustive pattern matching *)
    let head = tail.next in
    if head == tail then
      q.tail <- None
    else
      tail.next <- head.next;
    head.content
  let length q = q.length
end;;
```

Partage de type par contrainte

L'abstraction peut être une source de difficultés...

- Par masquage on peut rendre un type *abstrait*.
- A priori tous les types abstraits sont *différents*.
- Parfois, on souhaite spécifier que deux types abstraits sont le même de façon à les *partager* entre plusieurs modules.
- Le langage offre la possibilité d'exprimer des contraintes d'*égalité de types*.

Exemple (partage de types)

- Un module `M` avec un type abstrait `t`.

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

- On restreint la vue du module `M` de deux façons.

```
# module type S1 =
   sig
     type t
     val create : unit -> t
     val add : t -> unit
   end ;;

# module type S2 =
   sig
     type t
     val get : t -> int
   end ;;

# module M1 = (M:S1) ;;
module M1 : S1

# module M2 = (M:S2) ;;
module M2 : S2
```

- On restreint la vue du module `M` de deux façons.

```
# module type S1 =
   sig
     type t
     val create : unit -> t
     val add : t -> unit
   end ;;

# module type S2 =
   sig
     type t
     val get : t -> int
   end ;;

# module M1 = (M:S1) ;;
module M1 : S1

# module M2 = (M:S2) ;;
module M2 : S2
```

- Le problème est que les types `M1.t` et `M2.t` ne sont pas identifiés :

```
# let x= M1.create ();;
val x : M1.t = <abstr>

# M1.add x;;
- : unit = ()

# M2.get x;;
This expression has type M1.t but is here used with type M2.t
```

- On règle le problème avec des contraintes d'égalité

```
# module M1 = (M:S1 with type t = M.t) ;;
module M1 : sig
   type t = M.t
   val create : unit -> t
   val add : t -> unit
end

# module M2 = (M:S2 with type t = M.t) ;;
module M2 : sig
   type t = M.t
   val get : t -> int
end

# let x = M1.create() in M1.add x ;  M2.get x ;;
- : int = 1
```

Une solution alternative avec sous-modules

- On construit `M1` et `M2` comme *sous-modules* de `M`

```
# module M =
  (struct
     type t = int ref
     module M_hide =
       struct
         let create() = ref 0
         let add x = incr x
         let get x = if !x>0 then (decr x; 1) else failwith"Empty"
       end
     module M1 = M_hide
     module M2 = M_hide
   end
  :
   sig
     type t
     module M1 : sig  val create : unit -> t  val add : t -> unit end
     module M2 : sig  val get : t -> int  end
   end ) ;;
```

- Le type synthétisé est :

```
module M :
  sig
    type t
    module M1 : sig val create : unit -> t val add : t -> unit end
    module M2 : sig val get : t -> int end
  end
```

- Maintenant `M1` et `M2` font référence au même type abstrait.

```
# let x = M.M1.create() ;;
val x : M.t = <abstr>

# M.M1.add x ; M.M2.get x ;;
- : int = 1
```

- Le type synthétisé est :

```
module M :
  sig
    type t
    module M1 : sig val create : unit -> t val add : t -> unit end
    module M2 : sig val get : t -> int end
  end
```

- Maintenant `M1` et `M2` font référence au même type abstrait.

```
# let x = M.M1.create() ;;
val x : M.t = <abstr>

# M.M1.add x ; M.M2.get x ;;
- : int = 1
```

- Ce n'est pas "modulaire" (conception de `M1`, `M2` a posteriori impossible)

- Le type synthétisé est :

```
module M :
  sig
    type t
    module M1 : sig val create : unit -> t val add : t -> unit end
    module M2 : sig val get : t -> int end
  end
```

- Maintenant `M1` et `M2` font référence au même type abstrait.

```
# let x = M.M1.create() ;;
val x : M.t = <abstr>

# M.M1.add x ; M.M2.get x ;;
- : int = 1
```

- Ce n'est pas "modulaire" (conception de `M1`, `M2` a posteriori impossible)

- Avec les foncteurs (prochain argument) cette solution n'est plus viable

Inclusion de structures et signatures

Nous avons vu comment *restreindre* la vue d'une signature. Quid si on veut *élargir* une structure ou une signature ?

```
# module type S =
sig
  type t
  val x: t
  val f: t->t
end;;

# module type S1 =
sig
  include S           (* inclusion d'une signature *)
  val g: t->t
end;;

module type S1 = sig type t val x : t val f : t -> t val g : t -> t end
```

Un autre exemple avec Points et Cercles

```
module type POINT =
sig
  type point = float * float
  val mk_point: float * float -> point
  val x_coord:  point -> float
  val y_coord:  point -> float
  val move_p : point * float * float -> point
end;;

module type CIRCLE =
sig
  include POINT                          (* inclusion de la signature *)
  type circle
  val mk_circle: point * float -> circle
  val center:  circle -> point
  val radius:  circle -> float
  val move_c : circle * float * float -> circle
end;;
```

```
module Point: POINT =
struct
  type point = float * float
  let mk_point(x,y)  = (x,y)
  let x_coord(x,y) = x
  let y_coord(x,y) = y
  let move_p ((x,y),dx,dy) = (x+.dx,y+.dy)
end;;

module Circle: CIRCLE = struct
  include Point                          (* inclusion de la structure *)
  type circle = point * float
  let mk_circle (x,y) = (x,y)
  let center(x,y) = x
  let radius (x,y) = y
  let move_c(((x,y),r),dx,dy) = ((x+.dx,y+.dy),r)
end;;
```

Compilation séparée (1)

- Dans la programmation à grande échelle il convient de séparer un programme en plusieurs fichiers qui peuvent être compilés séparément.

Compilation séparée (1)

- Dans la programmation à grande échelle il convient de séparer un programme en plusieurs fichiers qui peuvent être compilés séparément.
- En OCAML, *unité de compilation = deux fichiers* :
  - le fichier implémentation `nom.ml` (= contenu d'une structure)
  - le fichier interface `nom.mli` (= contenu d'une signature)
  - Les deux fichiers sont équivalents à la déclaration

```
module Nom = (
    struct
      (* contenu de nom.ml *)
    end :
    sig
      (* contenu de nom.mli *)
    end
)
```

Compilation separeée (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
- Le fichiers `nom.ml` et `nom.mli` peuvent être compilés séparément avec l'option -c (compiler sans lier)

```
% ocamlc -c aux.mli                    produit  aux.cmi  code objet interface
% ocamlc -c aux.ml                     produit  aux.cmo  code objet implantation
% ocamlc -c main.mli                   produit  main.cmi code objet interface
% ocamlc -c main.ml                    produit  main.cmo code objet implantation
% ocamlc aux.cmo main.cmo -o main      linking
```

- Le programme est équivalent à :

```
module Aux: sig (* contenu de aux.mli *) end
          = struct (* contenu de aux.ml *) end;;
module Main: sig (* contenu de main.mli *) end
          = struct (* contenu de main.ml *) end;;
```

En particulier `Main` peut faire référence aux définitions dans l'interface de `Aux`, mais `Aux` ne peut pas faire référence à `Main`.

- Depuis la version 3.07 de OCaml il est possible de définir des structures et des signatures récursives par la syntaxe :

  ```
  module rec ... and ...
  ```

  avec des restrictions pour assurer la terminaison :

  - Tout cycle de dépendance doit passer par au moins un module “safe".
  - Un module est “safe” si tout valeur défini dans le module est une fonction
  - L’évaluation démarre en construisant les modules “safe" dont les valeurs sont initialisées à `fun _ -> raise Undefined_recursive_module`.

```
module rec A : sig
                 type t = Leaf of string | Node of ASet.t
                 val compare: t -> t -> int
               end
             = struct
                 type t = Leaf of string | Node of ASet.t
                 let compare t1 t2 =
                   match (t1, t2) with
                     (Leaf s1, Leaf s2) -> Pervasives.compare s1 s2
                   | (Leaf _, Node _) -> 1
                   | (Node _, Leaf _) -> -1
                   | (Node n1, Node n2) -> ASet.compare n1 n2
               end
    and ASet : Set.S with type elt = A.t
             = Set.Make(A)
```

On peut lui donner la spécification suivante

```
module rec A : sig
                 type t = Leaf of string | Node of ASet.t
                 val compare: t -> t -> int
               end
               and ASet : Set.S with type elt = A.t
```

Outline

Foncteurs

- Les *foncteurs* sont des fonctions des structures dans des structures.
- Ils sont utilisés pour exprimer une structure qui dépend d'une autre structure.
- Comme pour les fonctions, on écrit une fois un code qui pourra être utilisé plusieurs fois.

Exemple

- On définit la signature d'un *groupe*.

```
# module type GROUPE =
sig
  type g
  val e: g
  val comp: g*g -> g
  val inv: g -> g
end;;
```

Exemple

- On définit la signature d'un *groupe*.

```
# module type GROUPE =
sig
  type g
  val e: g
  val comp: g*g -> g
  val inv: g -> g
end;;
```

- On définit la construction de carré d'un groupe comme un 'foncteur' des groupes dans les groupes.

```
# module Square (Gr: GROUPE) =
  ( struct
     type g = Gr.g * Gr.g
     let e = (Gr.e,Gr.e)
     let comp ((a,b),(c,d)) = (Gr.comp(a,c),Gr.comp(b,d))
     let inv (a,b) = (Gr.inv(a),Gr.inv(b))
    end : GROUPE );;
module Square : functor (Gr : GROUPE) -> GROUPE
```

- On peut construire la structure GROUPE des entiers.

```
# module Zeta: GROUPE =
  struct
    type g = int
    let e = 0
    let comp (n,m) = n+m
    let inv (n) = -n
  end;;
```

- On peut construire la structure GROUPE des entiers.

```
# module Zeta: GROUPE =
  struct
    type g = int
    let e = 0
    let comp (n,m) = n+m
    let inv (n) = -n
  end;;
```

- et générer le groupe carré par application.

```
# module SquareZeta = Square(Zeta) ;;
module SquareZeta :
  sig
    type g = Square(Zeta).g
    val e : g
    val comp : g * g -> g
    val inv : g -> g
  end
```

NB Ici le type dans le résultat est *abstrait*.

Un autre exemple : mots ordonnés

- On déclare une signature *type ordonné*.

```
type comparison = Less | Equal | Greater;;

module type ORDERED_TYPE =
    sig
     type t
     val compare: t -> t -> comparison
    end;;
```

- On déclare un foncteur Set qui est paramétré sur un type ordonné.

```
module Set (Elt: ORDERED_TYPE) =
  struct
    type element = Elt.t
    type set = element list
    let empty = []
    let rec add x s =
      match s with
        [] -> [x]
      | hd::tl ->
         match Elt.compare x hd with
           Equal   -> s          (* x is already in s *)
         | Less    -> x :: s     (* x is smaller than all elmts of s *)
         | Greater -> hd :: add x tl
    let rec member x s =
      match s with
        [] -> false
      | hd::tl ->
          match Elt.compare x hd with
            Equal   -> true      (* x belongs to s *)
          | Less    -> false     (* x is smaller than all elmts of s *)
          | Greater -> member x tl
  end;;
```

Le type inferé est :

```
module Set :
functor (Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set = element list
      val empty : 'a list
      val add : Elt.t -> Elt.t list -> Elt.t list
      val member : Elt.t -> Elt.t list -> bool
    end
```

- On construit la structure *mots ordonnés*.

```
# module OrderedString =
    struct
      type t = string
      let compare x y =
            if x = y then Equal
            else if x < y then Less
            else Greater
    end;;
module OrderedString :
  sig type t = string val compare : 'a -> 'a -> comparison end
```

- On dérive par *application* la structure *ensembles de mots*

```
# module StringSet = Set(OrderedString);;
module StringSet :
  sig
    type element = OrderedString.t
    type set = element list
    val empty : 'a list
    val add : OrderedString.t -> OrderedString.t list -> OrderedString
    val member : OrderedString.t -> OrderedString.t list -> bool
  end

# StringSet.member "bar" (StringSet.add "foo" StringSet.empty);;
- : bool = false
```

Foncteurs et abstraction

- On souhaite cacher le fait que les ensembles sont représentés par des listes.

Foncteurs et abstraction

- On souhaite cacher le fait que les ensembles sont représentés par des listes.
- On déclare une signature *de foncteur*.

```
# module type SETFUNCTOR =
  functor (Elt: ORDERED_TYPE) ->
    sig
      type element = Elt.t      (* concrete *)
      type set                  (* abstract *)
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end;;
```

- On utilise la signature pour créer une vue abstraite de Set.

```
# module AbstractSet = (Set : SETFUNCTOR);;
module AbstractSet : SETFUNCTOR

# module AbstractStringSet = AbstractSet(OrderedString);;
module AbstractStringSet :
   sig
     type element = OrderedString.t
     type set = AbstractSet(OrderedString).set   ←list n'est plus visible!
     val empty : set
     val add : element -> set -> set
     val member : element -> set -> bool
   end

#AbstractStringSet.add "gee" AbstractStringSet.empty;;
 - : AbstractStringSet.set = <abstr>
```

Foncteurs et abstraction (suite)

On considère un ordre non-standard sur le mots (on ne distingue pas lettres majuscules et minuscules).

```
# module NoCaseString =
  struct
    type t = string
    let compare s1 s2 =
      OrderedString.compare (String.lowercase s1) (String.lowercase s2)
  end;;
module NoCaseString :
    sig type t = string val compare : string -> string -> comparison end
```

Foncteurs et abstraction (suite)

On considère un ordre non-standard sur le mots (on ne distingue pas lettres majuscules et minuscules).

```
# module NoCaseString =
  struct
    type t = string
    let compare s1 s2 =
      OrderedString.compare (String.lowercase s1) (String.lowercase s2)
  end;;
module NoCaseString :
    sig type t = string val compare : string -> string -> comparison end
```

On utilise le foncteur `AbstractSet` pour construire des ensembles de mots dont le type de representation est abstrait

```
# module NoCaseStringSet = AbstractSet(NoCaseString);;
module NoCaseStringSet :
    sig
      type element = NoCaseString.t
      type set = AbstractSet(NoCaseString).set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end
```

```
# NoCaseStringSet.add "FOO" AbstractStringSet.empty;;
  This expression has type
    AbstractStringSet.set = AbstractSet(OrderedString).set
  but is here used with type
    NoCaseStringSet.set = AbstractSet(NoCaseString).set
```

Les types AbstractStringSet.set et NoCaseStringSet.set sont *incompatibles*

```
# NoCaseStringSet.add "FOO" AbstractStringSet.empty;;
  This expression has type
    AbstractStringSet.set = AbstractSet(OrderedString).set
  but is here used with type
    NoCaseStringSet.set = AbstractSet(NoCaseString).set
```

Les types AbstractStringSet.set et NoCaseStringSet.set sont *incompatibles*

Nota Bene

Ceci est *souhaitable*. Par exemple, l'union sur AbstractStringSet est différente de l'union sur NoCaseStringSet.set.

Foncteurs et Contraintes

On nomme SET la signature de la structure rendue par le foncteur AbstractSet.

```
# module type SET = sig
    type element
    type set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end;;
```

Foncteurs et Contraintes

On nomme SET la signature de la structure rendue par le foncteur AbstractSet.

```
# module type SET = sig
    type element
    type set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end;;
```

On pourrait penser d'utiliser SET pour abstraire le foncteur Set

```
# module WrongSet = (Set : functor(Elt: ORDERED_TYPE) -> SET);;
module WrongSet : functor (Elt : ORDERED_TYPE) -> SET

# module WrongStringSet = WrongSet(OrderedString);;
module WrongStringSet : sig
    type element = WrongSet(OrderedString).element
    type set = WrongSet(OrderedString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
end

# WrongStringSet.add "gee" WrongStringSet.empty;;
This expression has type string but is here used with type
  WrongStringSet.element = WrongSet(OrderedString).element
```

Le problème est que `SET` spécifie le type des éléments de façon abstraite.
Ainsi `WrongStringSet.element` n'est pas le même type que `string`.
Pour surmonter cette difficulté on doit ajouter des *contraintes* :

```
# module AbstractSet =
    (Set : functor(Elt: ORDERED_TYPE) -> (SET with type element = Elt.t));;
module AbstractSet :
    functor (Elt : ORDERED_TYPE) ->
      sig
        type element = Elt.t
        type set
        val empty : set
        val add : element -> set -> set
        val member : element -> set -> bool
      end
```

Exercice (fonctions polymorphes et foncteurs)

- On veut définir une opération de tri polymorphe
  `tri : 'a list -> 'a list`
- On a besoin d'une opération de comparaison
  `lesseq: 'a -> a' -> bool`
- On est donc obligé de définir
  `tri : 'a list ->  ('a -> 'a -> bool) -> 'a list`

Exercice : Proposez une solution alternative selon le schéma suivant :

1. On définit une signature ORDTYPE 'type ordonné'.
2. On définit un foncteur qui prend en paramètre une structure 'type ordonné' et produit une structure avec une fonction de tri pour le type ordonné en question.

Une solution

```
(* Une signature pour les types ordonnés *)
module type ORDTYPE = sig
  type t
  val lesseq: t -> t -> bool
end ;;

(* Un module parametrique pour faire le tri par insertion *)
module Sort (OrdType : ORDTYPE ) = struct
  type t = OrdType.t
  let rec insert x l =  match l with
    [] -> [x] | y::l1 -> if OrdType.lesseq x y
                         then x::y::l1 else y:: insert x l1
  let rec isort l = match l with
    [] -> []  | y::l1 -> insert y  (isort l1)
end;;

(* Une structure de paires d'entiers avec ordre lexicographique *)
module OrdIntPair = struct
  type t = int * int
  let lesseq (x1,x2) (y1,y2) =
  if x1 <= y1 then true else (if x1=y1 then x2<= y2 else false)
end;;

(* Définition d'une structure Sort(OrdIntPair) *)
module S = Sort(OrdIntPair);;
```

```
(* Une liste de couples d'entiers *)

 # let l = [(2,4); (3,2); (1,5)];;
 val l : (int * int) list = [(2, 4); (3, 2); (1, 5)]

(* On utilise la structure pour trier la liste d'entiers *)

 # S.isort l;;
  - : OrdIntPair.t list = [(1, 5); (2, 4); (3, 2)]


(* La fonction insert est aussi visible *)

 # S.insert (4,5) (S.isort l);;
 - : OrdIntPair.t list = [(1, 5); (2, 4); (3, 2); (4, 5)] *)
```

```
(* Pour la cacher on definit *)

module Sort (OrdType : ORDTYPE ) = (
struct
  type t = OrdType.t
  let rec insert x l =  match l with
     [] -> [x] | y::l1 -> if OrdType.lesseq x y then x::y::l1
                          else y:: insert x l1
  let rec isort l = match l with
     [] -> []  | y::l1 -> insert y  (isort l1)
end
:
sig
  val isort : OrdType.t list -> OrdType.t list
end);;
```

Lectures conseillées

- J. Mitchell. Concepts in programming languages, chpt 9, Data abstraction and Modularity.

  Pour une perspective historique sur les notions d'abstraction et de modularité.

- E. Chailloux et al. Objective OCAML, chapitre 14, Programmation modulaire (en ligne).

  Pour les détails sur les modules en OCAML (ces transparents sont tirés de ce livre).

Lectures avancées

- D. Mac Queen. Modules for standard ML. ACM POPL, 1984.

  On décrit la conception du système de modules de ML.

- Claudio Russo. Recursive structures for standard ML ACM ICFP, 2001.
- D. Dreyer, K. Crary, R. Harper. A type system for higher-order modules. ACM POPL, 2003.

  We present a type theory for higher-order modules that accounts for many central issues in module system design, including translucency, applicativity, generativity, and modules as first-class values (. . .)