

# Modules et types de données abstraits

CSI 3520

Amy Felty

University of Ottawa

# Développement de logiciels—la réalité

On connaît rarement les *bons* algorithmes ou les structures de données *appropriées* lorsqu'on commence un grand projet logiciel.

- Pour implémenter un moteur de recherche, quelles structures de données et quels algorithmes faut-il utiliser pour créer l'index? Pour implémenter l'évaluateur de requête?

La réalité est que *l'on modifie souvent le code* après la construction d'un prototype.

- Souvent, on ne sait même pas ce que *l'utilisateur souhaite* (exigences du logiciel) avant de tester un prototype.
- Souvent, on ne sait pas où sont les *problèmes de performances* avant de pouvoir exécuter le logiciel sur des test réalistes.
- Parfois, on veut juste changer de conception (proposer des algorithmes plus *simples*, une architecture plus *simple*) à une étape ultérieure du développement du logiciel.

# Planification des modifications

Étant donné que le logiciel va changer, comment écrire le code pour que les changements soient plus faciles?

La règle principale: utiliser *l'abstraction de données* et *l'abstraction d'algorithme*.

- Ne pas utiliser de *représentations concrètes* fournies par le langage de programmation.
- Utiliser *des abstractions de haut niveau* qui correspondent au domaine du problème.
- Implémenter les abstractions en utilisant une *interface bien définie*.
- *Modifier les implantations* des abstractions selon vos besoins.
- Effectuer des tâches de développement *en parallèle*.

# Types de données abstraits

En anglais: Abstract Data Type (ADT)



Barbara Liskov  
Professeur adjoint, MIT  
1973

Inventé le langage CLU  
qui impose l'abstraction des données



Barbara Liskov  
Professeur titulaire, MIT  
Turing Award 2008

“For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing.”

# Abstraction imposée par le langage

***Règle de base: utilisez les fonctionnalités du langage pour appliquer une abstraction.***

- **Loi de Murphy sur l'abstraction de données:**
  - Ce qui n'est pas imposé sera brisé à un certain moment par un client
- **Les systèmes de modules sont conçus exactement à cet effet**
  - Ils révèlent très peu d'informations sur l'implantation.
  - Ils offrent une flexibilité maximale pour les changements.
  - Il peut y avoir de gros gains plus tard.
- **Comme toutes les règles de conception, faites une exception si nécessaire**
  - Reconnaissez quand une règle pose plus de problèmes qu'elle n'en résout
- **Le système de modules de ML est particulièrement efficace.**

# Création de types de données abstraits dans OCaml

Utilisez les modules OCaml pour créer de nouveaux types de données abstraits!

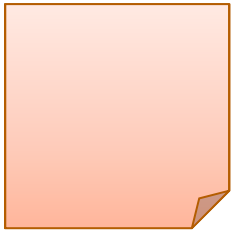
- **signature**: une interface
  - spécifie le(s) type(s) de données abstrait(s) sans spécifier leur implantation
  - spécifie les opérations sur les types de données abstraits
- **structure**: une implantation
  - une collection de définitions de type et de valeur
  - une implantation « correspond » à une interface ou « satisfait » une interface
    - Ceci fournit une notion de sous-typage
- **foncteur**: un module paramétré
  - une fonction dont l'entrée est des modules, renvoie un module
  - permet de factoriser et de réutiliser des modules

# Modules simples

## Convention OCaml :

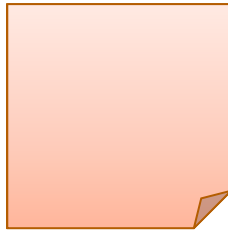
- fichier Name.ml est une *structure* qui implémente un module nommé **Name**
- fichier Name.mli est une *signature* pour le module nommé **Name**
  - s'il n'y a pas de fichier Name.mli, OCaml déduit la signature (un défaut)
- D'autres modules, par exemple ClientA ou ClientB peuvent:
  - utiliser la *notation « point »* pour faire référence au contenu de Name, par exemple: Name.val
  - **open Name**: pour avoir accès à tous les éléments de Name
    - l'ouverture d'un module peut introduire beaucoup de noms dans l'espace de nommage.

Signature



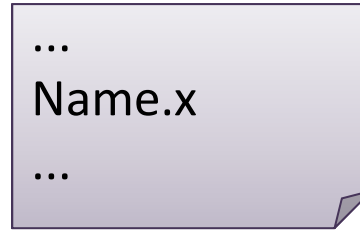
Name.mli

Structure

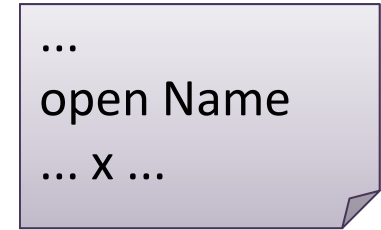


Name.ml

*Faites attention quand vous ouvrez des modules!*



ClientA.ml



ClientB.ml

# À première vue: modules OCaml = modules C?

## C utilise :

- Fichiers .h (des signatures) similaires aux fichiers .mli?
- Fichiers .c (des structures) similaires aux fichiers .ml?

## Mais ML a aussi :

- contrôle plus strict sur les types de données abstraits
  - définir des types de données abstraits, transparents ou translucides dans les signatures, c'est-à-dire ne donner aucune, donner toutes ou donner une partie des informations sur les types aux clients
- plus de structure
  - les modules peuvent être définis dans des autres modules, c'est-à-dire que les signatures et les structures peuvent être définies dans des fichiers
- plus de réutilisation
  - plusieurs modules peuvent satisfaire la même interface
  - le même module peut satisfaire plusieurs interfaces
  - les modules peuvent être des arguments à d'autres modules (foncteurs)
- beaucoup de fonctionnalités: des modules dynamiques et de première classe!



# À première vue: modules OCaml = modules C?

## C utilise :

- Fichiers .h (des signatures) similaires aux fichiers .mli?
- Fichiers .c (des structures) similaires aux fichiers .ml?

## Mais ML a aussi :

- contrôle plus strict sur les types et données abstraites
  - définir des types de données et des opérations dans les signatures, ou donner une partie de la mise en œuvre
- plus de flexibilité
  - les modules peuvent être utilisés de différentes manières, à dire que les signatures peuvent être étendues
- plus de fonctionnalités
  - plusieurs modules peuvent implémenter la même interface
  - le même module peut satisfaire plusieurs interfaces
  - les modules peuvent être des arguments à d'autres modules (foncteurs)
- beaucoup de fonctionnalités: des modules dynamiques et de première classe!

**ML = Gagnant!**

## Un exemple de signature

```
module type INT_STACK =  
  sig  
    type t  
    val empty : unit -> t  
    val push   : int -> t -> t  
    val is_empty : t -> bool  
    val pop    : t -> t  
    val top    : t -> int option  
  end
```

## Un exemple de signature

```
module type INT_STACK =  
  sig  
    type t  
    val empty : unit -> t  
    val push  : int -> t -> t  
    val is_empty : t -> bool  
    val pop   : t -> t  
    val top   : t -> int option  
  end
```

*convention:* quand  
il y a un seul type  
dans le module,  
utilisez le nom **t**.

les clients utilisent  
**Stack.t**

## Un exemple de signature

```
module type INT_STACK =  
  sig  
    type t  
    val empty : unit -> t  
    val push  : int -> t -> t  
    val is_empty : t -> bool  
    val pop   : t -> t  
    val top   : t -> int option  
  end
```

« empty » et  
« push » sont des  
**constructeurs**  
abstrait: fonctions  
qui construisent ce  
type de données  
abstrait.

## Un exemple de signature

```
module type INT_STACK =  
  sig  
    type t  
    val empty : unit -> t  
    val push  : int -> t -> t  
    val is_empty : t -> bool  
    val pop : t -> t  
    val top : t -> int  
  end
```

« is\_empty » est un **observateur** – utile pour déterminer les propriétés du type de données abstrait

## Un exemple de signature

```
module type INT_STACK =  
  sig  
    type t  
    val empty : unit -> t  
    val push  : int -> t -> t  
    val is_empty : t -> bool  
    val pop : t -> t  
    val top : t -> int option  
  end
```

« pop » est parfois  
appelé un  
***mutateur*** (mais  
cela ne change pas  
vraiment l'entrée)

## Un exemple de signature

```
module type INT_STACK =  
  sig  
    type t  
    val empty : unit -> t  
    val push  : int -> t -> t  
    val is_empty : t -> bool  
    val pop   : t -> t  
    val top   : t -> int option  
  end
```

« top » est également un *observateur*, dans ce contexte fonctionnel, car il ne modifie pas la pile

# Mettez des commentaires dans les signatures!

```
module type INT_STACK =
  sig
    type t
    (* créer une pile vide *)
    val empty : unit -> t

    (* empiler un élément sur la pile *)
    val push : int -> t -> t

    (* renvoie vrai si la pile est vide, faux sinon *)
    val is_empty : t -> bool

    (* dépile la tête de la pile;
       renvoie une pile vide si la pile est vide *)
    val pop : t -> t

    (* renvoie la tête de la pile;
       renvoie None si la pile est vide *)
    val top : t -> int option
  end
```



## Commentaires dans les signatures

- Les commentaires de signature sont pour les clients du module
  - Ils expliquent ce que chaque fonction doit faire
    - comment elle traite les valeurs abstraites (piles)
  - ***pas*** comment elle traite des valeurs concrètes
  - Ils ne révèlent pas les détails d'implantation qui devraient être abstraits
- Ne répétez pas les commentaires de la signature dans les structures
  - les commentaires deviendront obsolètes à un endroit ou à un autre
  - une extension de la règle générale: il ne faut pas copier du code
- Mettez les commentaires d'implantation dans les structures
  - les commentaires sur les invariants d'implantation doivent être cachés du client
  - incluez des commentaires sur les fonctions auxiliaires

## Un exemple de structure

```
module ListIntStack : INT_STACK =  
  struct  
    type t = int list  
    let empty () : t = []  
    let push (i:int) (s:t) : t = i::s  
    let is_empty (s:t) =  
      match s with  
        | [] -> true  
        | _::_ -> false  
    let pop (s:t) : t =  
      match s with  
        | [] -> []  
        | _::tl -> tl  
    let top (s:t) : int option =  
      match s with  
        | [] -> None  
        | h::_ -> Some h  
  end
```

# Un exemple de structure

```
module ListIntStack : INT_STACK =  
  struct  
    type t = int list  
    let empty () : t = []  
    let push (i:int) (s:t) : t =  
    let is_empty (s:t) =  
      match s with  
      | [] -> true  
      | _::_ -> false  
    let pop (s:t) : t =  
      match s with  
      | [] -> []  
      | _::tl -> tl  
    let top (s:t) : int option =  
      match s with  
      | [] -> None  
      | h::_ -> Some h  
  end
```

A l'intérieur du module, on connaît **le type concret** qui implémente le type de données abstrait.

# Un exemple de structure

```
module ListIntStack : INT_STACK =  
  struct  
    type t = int list  
    let empty () : t = []  
    let push (i:int) (s:t) : t = ...  
    let is_empty (s:t) =  
      match s with  
        | [] -> true  
        | _::_ -> false  
    let pop (s:t) : t =  
      match s with  
        | [] -> []  
        | _::tl -> tl  
    let top (s:t) : int option =  
      match s with  
        | [] -> None  
        | h::_ -> Some h  
  end
```

L'utilisation de l'interface du module INT\_STACK cache la façon dont les piles sont représentées. Le client ne peut pas utiliser le fait que les piles sont des listes.

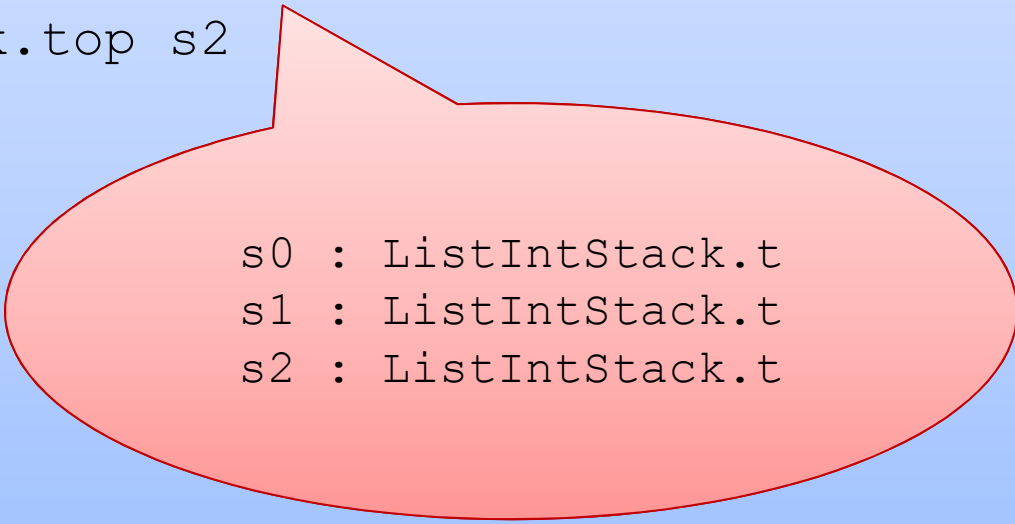
## Un exemple de client

```
module ListIntStack : INT_STACK =  
  struct  
    ...  
  end  
  
let s0 = ListIntStack.empty ()  
let s1 = ListIntStack.push 3 s0  
let s2 = ListIntStack.push 4 s1  
let i = ListIntStack.top s2
```

## Un exemple de client

```
module ListIntStack : INT_STACK =  
  struct  
    ...  
  end
```

```
let s0 = ListIntStack.empty ()  
let s1 = ListIntStack.push 3 s0  
let s2 = ListIntStack.push 4 s1  
let i = ListIntStack.top s2
```



```
s0 : ListIntStack.t  
s1 : ListIntStack.t  
s2 : ListIntStack.t
```

## Un exemple de client

```
module ListIntStack : INT_STACK =  
  struct  
    ...  
  end  
  
let s0 = ListIntStack.empty ()  
let s1 = ListIntStack.push 3 s0  
let s2 = ListIntStack.push 4 s1  
let i = ListIntStack.top s2  
      (* i : int option = Some 4 *)
```

## Un exemple de client

```
module ListIntStack : INT_STACK =  
  struct  
    ...  
  end  
  
let s0 = ListIntStack.empty ()  
let s1 = ListIntStack.push 3 s0  
let s2 = ListIntStack.push 4 s1  
let i = ListIntStack.top s2  
      (* i : int option = Some 4 *)  
let j = ListIntStack.top (ListIntStack.pop s2)  
      (* j : int option = Some 3 *)
```



## Un exemple de client

```
module ListIntStack : INT_STACK =  
  struct  
    ...  
  end  
  
let s0 = ListIntStack.empty ()  
let s1 = ListIntStack.push 3 s0  
let s2 = ListIntStack.push 4 s1  
let i = ListIntStack.top s2  
      (* i : int option = Some 4 *)  
let j = ListIntStack.top (ListIntStack.pop s2)  
      (* j : int option = Some 3 *)  
open ListIntStack
```

## Un exemple de client

```
module ListIntStack : INT_STACK =  
  struct  
    ...  
  end  
  
let s0 = ListIntStack.empty ()  
let s1 = ListIntStack.push 3 s0  
let s2 = ListIntStack.push 4 s1  
let i = ListIntStack.top s2  
      (* i : int option = Some 4 *)  
let j = ListIntStack.top (ListIntStack.pop s2)  
      (* j : int option = Some 3 *)  
open ListIntStack  
let k = top (pop (pop s2))  
      (* k : int option = None *)
```

## Un exemple de client

```
module type INT_STACK =  
  sig  
    type t  
    val push  : int -> t -> t  
    ...  
  
module ListIntStack : INT_STACK  
  
let s2 = ListIntStack.push 4 s1  
...  
let l  = List.rev s2
```

Notez que le client n'est pas autorisé à savoir que la pile est une liste.

**Error: This expression has type ListIntStack.t but an expression was expected of type 'a list.**

# Un exemple de structure

```
module ListIntStack (* : INT_STACK *) =  
  struct  
    type t = int list  
    let empty () : t = []  
    let push (i:int) (s:t) = i::s  
    let is_empty (s:t) =  
      match s with  
        | [] -> true  
        | _::_ -> false  
    exception EmptyStack  
    let pop (s:t) =  
      match s with  
        | [] -> []  
        | _::tl -> tl  
    let top (s:t) =  
      match s with  
        | [] -> None  
        | h::_ -> Some h  
  end
```

Lors du débogage, on peut utiliser un commentaire pour la signature afin d'accéder au contenu du module.

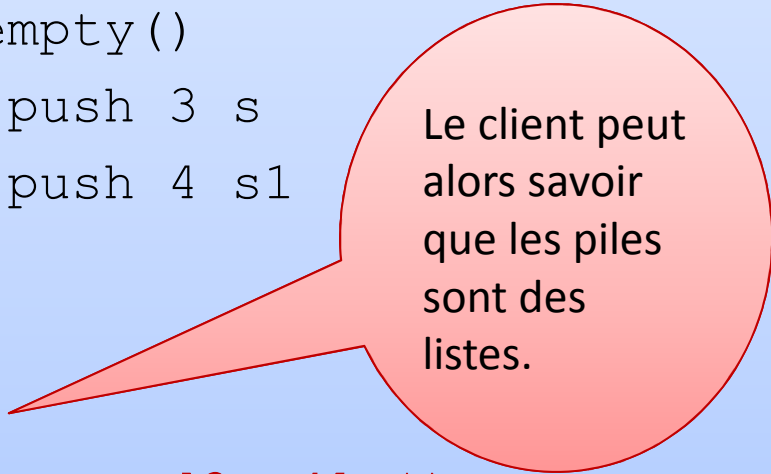
## Le client sans la signature

```
module ListIntStack (* : INT_STACK *) =  
  struct  
    ...  
  end
```

```
let s = ListIntStack.empty()  
let s1 = ListIntStack.push 3 s  
let s2 = ListIntStack.push 4 s1
```

...

```
let l = List.rev s2  
      (* l : int list = [3; 4] *)
```



Le client peut  
alors savoir  
que les piles  
sont des  
listes.

# Un exemple de structure

```
module ListIntStack : INT_STACK =  
  struct  
    type t = int list  
    let empty () : t = []  
    let push (i:int) (s:t) = i::s  
    let is_empty (s:t) =  
      match s with  
      | [] -> true  
      | _::_ -> false  
    exception EmptyStack  
    let pop (s:t) =  
      match s with  
      | [] -> []  
      | _::tl -> tl  
    let top (s:t) =  
      match s with  
      | [] -> None  
      | h::_ -> Some h  
  end
```

Quand on utilise la signature, on restreint l'accès du client aux informations contenues dans la signature (ce qui ne révèle pas que `stack = int list`.) Les clients ne peuvent donc utiliser **que** les opérations de pile sur une valeur de pile (pas les opérations de liste).

# Un exemple de structure

```
module type INT_STACK =  
  sig  
    type stack  
    ...  
  
    val inspect : stack -> int list  
    val run_unit_tests : unit -> unit  
  
  end  
  
module ListIntStack : INT_STACK =  
  struct  
    type stack = int list  
    ...  
  
    let inspect (s:stack) : int list = s  
    let run_unit_tests () : unit = ...  
  
  end
```

Une autre technique:

Ajoutez des fonctions de test à la signature.

Une autre option consiste à: 2 signatures, une pour les tests et une pour le reste du code)

## Rappelez la signature pour les piles d'entiers

```
module type INT_STACK =  
  sig  
    type t  
    val empty : unit -> t  
    val push  : int -> t -> t  
    val is_empty : t -> bool  
    val pop : t -> t  
    val top : t -> int option  
  end
```



# Une signature pour les piles polymorphes

```
module type INT_STACK =  
  sig  
    type t  
    val empty : unit -> t  
    val push  : int -> t -> t  
    val is_empty : t -> bool  
    val pop : t -> t  
    val top  
  end
```

```
module type STACK =  
  sig  
    type 'a stack  
    val empty : unit -> 'a stack  
    val push  : 'a -> 'a stack -> 'a stack  
    val is_empty : 'a stack -> bool  
    val pop : 'a stack -> 'a stack  
    val top : 'a stack -> 'a  
  end
```

# Une implantation

```
module ListStack : STACK =  
  struct  
    type `a stack = `a list  
    let empty() : `a stack = []  
    let push (x:`a) (s:`a stack) : `a stack = x::s  
    let is_empty(s:`a stack) =  
      match s with  
      | [] -> true  
      | _::_ -> false  
    let pop (s:`a stack) : `a stack =  
      match s with  
      | [] -> []  
      | _::tl -> tl  
    let top (s:`a stack) : `a option =  
      match s with  
      | [] -> None  
      | h::_ -> Some h  
  end
```

# Résumé

- On est souvent tenté de supprimer la barrière d'abstraction.
  - Par exemple, on veut imprimer un ensemble et on utilise une fonction sur les listes.
- Mais l'objectif principal de la barrière est de soutenir les futurs changements.
  - par exemple, changer un invariant sur des données non triées en un autre sur des données triées.
  - Ou changer une implantation qui utilise des listes en une qui utilise des arbres équilibrés.
- Dans de nombreux langages, il est possible d'avoir des fuites à travers la barrière d'abstraction.
  - Les «bons» clients ne devraient pas en profiter.
  - mais ils finissent toujours par le faire,
  - donc les modifications de code doivent continuer à permettre ces fuites, sinon le code client ne fonctionnera plus

# Points clés

Les mécanismes d'OCaml incluent:

- *signatures* (interfaces)
- *structures* (implantations)
- *functors* (des fonction dont l'entrées sont des modules et qui renvoie un module)

On peut utiliser le système de modules

- pour fournir les espaces de nommages
- pour *cacher des informations* (types, définitions de valeurs locales)
- pour réutilisation de code (via des foncteurs, des interfaces réutilisables, des modules réutilisables)

Le masquage d'informations permet la conception de types de données *abstraits* et d'algorithmes *abstraits*.

- ensembles au lieu de listes, tableaux ou arbres
- documents au lieu de chaînes
- plus il y a de caché, plus il est facile de remplacer l'implantation
- utiliser les fonctionnalités du langage pour implémenter le masquage d'informations
  - il est toujours possible d'ignorer les invariants dans les commentaires
  - utilisez le contrôle de type pour vous garantir une forte abstraction