Programmation Avancée (INF441)

Composants (paramétrés)

François Pottier

10 mai 2016

Pourquoi paramétrer ?

En paramétrant un composant,

- on évite de fixer trop tôt certains choix ou certains détails ;
- on rend ainsi ce composant plus général, plus ré-utilisable.

Quoi paramétrer ? Par quoi paramétrer ?

Lorsqu'on parle de « paramétrer »,

- que peut-on paramétrer,
- et par quoi ?

Types et valeurs

En Java et en OCaml, nous avons manipulé deux sortes d'entités :

- des types
  - ils n'existent pas à l'exécution
  - ils offrent une documentation vérifiée par le compilateur
- des valeurs
  - entiers, paires, arbres, fonctions, ...
  - elles existent à l'exécution
  - elles sont décrites par des types

On peut paramétrer un type ou une valeur par un type ou un valeur, d'où 4 combinaisons...

Types paramétrés et types dépendants

Un type peut être paramétré...

... par un type :

- `List<X>` est le type des listes d'éléments de type `X`

... par une valeur ?

- `List<5, X>` pourrait être le type des listes de longueur 5
- ces types dépendants n'existent pas dans Java ni OCaml

Fonctions polymorphes et d'ordre supérieur

Une valeur peut être paramétrée...

... par un type :

- `List.length` a le type `'a list -> int` pour tout type `'a`
- on appelle cela une fonction polymorphe

... par une valeur :

- on appelle cela une fonction
- c'est particulièrement intéressant si le paramètre est une fonction
- `List.sort` a le type `('a -> 'a -> int) -> 'a list -> 'a list`
- on appelle cela une fonction d'ordre supérieur

Des types et valeurs aux composants

J'ai parlé plusieurs fois déjà de « composant ».

Pour moi, un composant regroupe :

- un ou plusieurs types, souvent abstraits
- une ou plusieurs valeurs, souvent des fonctions, ou « opérations »

Quelques exemples de composants :

- « une implémentation des ensembles »
- « une implémentation des files de priorité »

Des types et valeurs aux composants

Lorsqu'on veut paramétrer, on pense souvent en termes de composants, et non pas en termes de types et de valeurs individuels.

Quelques exemples :

- Étant donné un composant « type ordonné », on peut définir un composant « ensembles », à l'aide d'arbres binaires de recherche
- Étant donné un composant « type doté d'un test d'égalité et d'une fonction de hachage », on peut définir un composant « ensembles », à l'aide de tables de hachage
- Étant donné un composant « files de priorité », on peut définir un composant « compression », suivant la technique de Huffman

Des types et valeurs aux composants

Le langage de programmation nous aide-t-il à penser en ces termes ?

Peut-on paramétrer un composant par un composant ?

Si non, paramétrer M types et valeurs par N types et valeurs est lourd...
Cf. le TD4, où :

- le type `'a t` est paramétré par `'a`
- la fonction `Hashset.find` est paramétrée par `'a`, `hash` et `eq`
- la fonction `Hashset.add` est paramétrée par `'a`, `hash` et `eq`

Cette description est de taille $M \times N$, alors que $M + N$ devrait suffire.

Modules et foncteurs

OCaml propose :

- des composants, appelés aussi structures ou modules ;
- des types de composants, ou signatures ;
- des composants paramétrés par des composants, ou foncteurs.

Java... n'a pas de « composants » au sens où je l'entends, car :

- une classe ou interface contient plusieurs méthodes / opérations,
- mais ne peut pas contenir de types.

Note historique

CLU (Liskov, 1973–77) a été le premier langage à :

- voir un composant comme (un) type (abstrait) muni d'opérations,
- permettre de paramétrer un composant par un composant.

Idées reprises ensuite par Ada (1983), Modula-2 (1977) et Modula-3 (198x), Standard ML (198x), Scala (2003)...

Modules / Structures

Un module regroupe des définitions de types et de valeurs.

Nous avons déjà construit des modules... Si dans `I.ml` on écrit :

```
type t = int
let compare (x : t) (y : t) : int =
  if x < y then -1 else if x > y then 1 else 0
```

Alors OCaml considère qu'on a défini un module nommé `I`.

Il contient le type `I.t` et la fonction `I.compare`.

Signatures

Une signature regroupe des définitions ou déclarations de types et des déclarations de valeurs.

Nous avons déjà construit des signatures... Si dans `I.mli` on écrit :

```
type t = int
val compare: t -> t -> int
```

Alors OCaml vérifie que le module `I` est conforme à cette signature.

À l'extérieur, on ne connaît de `I` que ce qui est dit dans cette signature.

Ainsi, si `t` est déclaré mais non défini dans `I.mli`, il devient abstrait :

```
type t
val compare: t -> t -> int
```

Vers plus de flexibilité

Jusqu'ici, un module = deux fichiers (`.ml` et `.mli`).

Cela rappelle Java 1.0, où une classe = un fichier (`.java`).

Cette convention est simple et utile mais trop rigide.

Pour que les composants (paramétrés) soient faciles à construire et à assembler, il faut que la notion de composant soit interne au langage.

C'est pourquoi OCaml prévoit aussi des moyens de construire structures et signatures dans le langage (dans un seul fichier)...

Modules / Structures

On s'autorise à définir un module `I` au beau milieu d'un fichier `A.ml` :

```
module I = struct
  type t = int
  let compare (x : t) (y : t) : int =
    if x < y then -1 else if x > y then 1 else 0
end
```

Dans la suite du fichier, on peut utiliser `I.t` et `I.compare`.

Vu depuis l'extérieur, ce module s'appelle `A.I`. On peut utiliser `A.I.t` et `A.I.compare`.

OCaml considère le module `I` comme imbriqué dans le module `A`.

Un module contient en fait des types, des valeurs, et des modules.

Attribution d'une signature à une structure

On peut attribuer une signature au module `I` :

```
module I : sig
  type t = int
  val compare: t -> t -> int
end = struct
  type t = int
  let compare (x : t) (y : t) : int =
    if x < y then -1 else if x > y then 1 else 0
end
```

La forme générale est `module I : <signature> = <structure>`.

Ici, cela n'apporte rien. OCaml pouvait inférer cette signature.

Attribution d'une signature à une structure

On peut aussi attribuer une signature plus abstraite au module `I` :

```
module I : sig
  type t (* an abstract type! *)
  val compare: t -> t -> int
end = struct
  type t = int
  let compare (x : t) (y : t) : int =
    if x < y then -1 else if x > y then 1 else 0
end
```

Dans ce cas, à l'extérieur, on ne sait plus que le type `I.t` est égal à `int`.

Ici, cela rend `I.compare` inutilisable.

Signatures nommées

On peut nommer une signature.

```
module type OrderedType = sig
  type t
  val compare: t -> t -> int
end
```

Tout module `M` qui satisfait cette signature est « un type muni d'une relation d'ordre », ou « un type ordonné ».

La forme générale est `module type S = <signature>`.

Attribution d'une signature nommée

En cas particulier, notre module `I` est « un type ordonné » :

```
module I : OrderedType = struct
  type t = int
  let compare (x : t) (y : t) : int =
    if x < y then -1 else if x > y then 1 else 0
end
```

Toujours la forme `module I : <signature> = <structure>`.

Comme précédemment, ceci rend le type `I.t` abstrait.

Ajout d'une équation à une signature

La signature plus précise, qui donne la définition du type `t` :

```
sig
  type t = int
  val compare: t -> t -> int
end
```

peut aussi s'écrire en ajoutant une équation à la signature `OrderedType` :

```
OrderedType with type t = int
```

On peut ajouter a posteriori une ou plusieurs équations à une signature, pour obtenir une signature plus précise.

Structures

On peut attribuer à `I` cette signature plus précise :

```
module I : (OrderedType with type t = int) = struct
  type t = int
  let compare (x : t) (y : t) : int =
    if x < y then -1 else if x > y then 1 else 0
end
```

Toujours la forme `module I : <signature> = <structure>`.

Vu de l'extérieur, le type `I.t` est concret, synonyme de `int`.

Structures

On peut définir d'autres « types ordonnés » :

```
module S : (OrderedType with type t = string) = struct
  type t = string
  let compare (s1 : t) (s2 : t) : int =
    ...
end
```

`S` admet la signature précise ci-dessus, et aussi la signature `OrderedType`.

Où en sommes-nous ?

Nous avons pu :

- nommer l'idée de « type ordonné », sous forme d'une signature ;
- définir plusieurs structures ou modules conformes à ce concept.

Où en sommes-nous ?

Nous allons maintenant pouvoir construire des modules paramétrés par un « type ordonné », par exemple :

- un module « files de priorité » ;
- un module « ensembles » à base d'arbres binaires de recherche ;

ou paramétrés par plusieurs « types ordonnés », par exemple :

- un « type ordonné » construit à l'aide de l'ordre lexicographique.

Foncteurs

OCaml appelle foncteur une fonction qui à un (ou plusieurs) modules associe un module.

On parle aussi de module paramétré par un (ou plusieurs) modules.

Définition d'un foncteur

Voici un foncteur qui attend deux modules `T1` et `T2` et renvoie un module.

```
module Lexico (T1 : OrderedType) (T2 : OrderedType) = struct
  type t = T1.t * T2.t
  let compare (x1, x2) (y1, y2) =
    let c1 = T1.compare x1 y1 in
    if c1 <> 0 then c1
    else T2.compare x2 y2
end
```

Les types `T1.t` et `T2.t` sont ici abstraits (inconnus).

Ce code a un sens quels que soient ces types.

Quel est le type de ce foncteur ?

Quelle est la signature de la structure produite par ce foncteur ?

On peut l'écrire ainsi :

```
sig
  type t = T1.t * T2.t
  val compare: (T1.t * T2.t) -> (T1.t * T2.t) -> int
end
```

Quel est le type de ce foncteur ?

Ou encore :

```
sig
  type t = T1.t * T2.t
  val compare: t -> t -> int
end
```

Ou encore :

```
OrderedType with type t = T1.t * T2.t
```

Déclaration d'un foncteur

Si `Lexico` était défini dans `A.ml`, alors dans `A.mli` on peut déclarer :

```
module Lexico (T1 : OrderedType) (T2 : OrderedType) :
  OrderedType with type t = T1.t * T2.t
```

Appliqué à deux « types ordonnés », il construit un « type ordonné ».

On précise que le support de ce dernier est le produit des supports.

Application d'un foncteur

Le foncteur `Lexico`, appliqué à deux structures, produit une structure :

```
module  I = struct type t = int    ... end (* comme avant *)
module  S = struct type t = string ... end (* comme avant *)
module IS = Lexico(I)(S)
```

La signature du module `IS` est

```
OrderedType with type t = I.t * S.t
```

ou encore :

```
OrderedType with type t = int * string
```

Le type `IS.t` est égal à `int * string`. (OCaml le sait.)

`IS.compare` compare des paires d'un entier et d'une chaîne de caractères.

Application d'un foncteur

Bien sûr, un même foncteur peut servir plusieurs fois :

```
module II = Lexico(I)(I)
```

La signature du module `II` est

```
OrderedType with type t = I.t * I.t
```

ou encore :

```
OrderedType with type t = int * int
```

`II.compare` compare des paires d'entiers.

Définition

Exemple tiré de `set.ml`, de la bibliothèque d'OCaml :

```
module Make (E : OrderedType) = struct
  type element = E.t
  type set = Empty | Node of set * element * set * int
  let empty : set = Empty
  let add (x: element) (s : set) : set = ...
  let mem (x : element) (s : set) : bool = ...
  let union (s1 : set) (s2 : set) : set = ...
end
```

Étant donné un « type ordonné » `E`, le foncteur `Set.Make` construit une implémentation des ensembles à base d'arbres binaires de recherche.

Déclaration

Dans `set.mli`, on trouve la déclaration suivante :

```
module Make (E : OrderedType) : sig
  type element = E.t
  type set
  val empty: set
  val add: element -> set -> set
  val mem: element -> set -> bool
  val union: set -> set -> set
end
```

Le type des éléments doit rester concret. (Pourquoi ?)

Le type des ensembles doit être abstrait. (Pourquoi ?)

Applications

Bien sûr, ce foncteur peut servir plusieurs fois.

Ensembles d'entiers, ensembles de chaînes de caractères :

```
module IntegerSet = Set.Make(I)
module StringSet = Set.Make(S)
```

Ensembles de paires d'entiers :

```
module IntegerPairSet = Set.Make(Lexico(I)(I))
```

Quelle est la signature du résultat ?

La signature du module `IntegerSet` est :

```
sig
  type element = I.t (* = int *)
  type set
  val empty: set
  val add: element -> set -> set
  val mem: element -> set -> bool
  val union: set -> set -> set
end
```

On a remplacé le paramètre formel `E` par l'argument effectif `I` dans la signature résultat de `Make`.

OCaml sait donc que `IntegerSet.element` = `I.t` = `int`.

Cependant, `IntegerSet.set` est un (nouveau) type abstrait.

Quelle est la signature du résultat ?

De même, la signature du module `StringSet` est :

```
sig
  type element = S.t (* = string *)
  type set
  val empty: set
  val add: element -> set -> set
  val mem: element -> set -> bool
  val union: set -> set -> set
end
```

OCaml sait donc que `String.element` = `S.t` = `string`.

Cependant, `StringSet.set` est un (nouveau) type abstrait.

Généralité

Chaque application du foncteur engendre un nouveau type abstrait.

En effet, si on écrit :

```
module Bar = Set.Make(Foo)
```

Alors le type `Bar.set` est considéré comme égal à aucun type connu auparavant, donc « nouveau ».

Ce phénomène est appelé généralité.

Intérêt de la générativité

Munissons le type des entiers de deux ordres différents :

```
module Increasing = struct
  type t = int
  let compare x y =
    if x < y then -1 else if x > y then 1 else 0
end
module Decreasing = struct
  type t = int
  let compare x y = Increasing.compare y x
end
```

Ces modules ont la même signature : `OrderedType with type t = int`.

Intérêt de la générativité

Construisons maintenant deux implémentations des ensembles d'entiers :

```
module IntegerSet1 = Set.Make(Increasing)
module IntegerSet2 = Set.Make(Decreasing)
```

Que dire des types `IntegerSet1.set` et `IntegerSet2.set` ?

OCaml les considère-t-il comme distincts ou non ?

Aimerions-nous qu'ils soient considérés comme distincts ou non ?

Intérêt de la générativité

OCaml les considère tous deux comme abstraits.

Donc, égaux à aucun autre type ; donc, distincts l'un de l'autre.

Et cela est souhaitable, en effet.

Sans cela, on pourrait mélanger des arbres qui respectent des relations d'ordre différentes.

Cela ne provoquerait pas de « plantage », mais ce serait une erreur : une violation de l'invariant des arbres binaires de recherche.

À retenir

Le typage statique ne sert pas seulement à éviter les « plantages », mais aussi à faire respecter des abstractions (Reynolds, 1983).

Et nos tables de hachage ?

Dans le TD4 :

- le type `'a t` est paramétré par `'a`
- la fonction `Hashset.find` est paramétrée par `'a`, `hash` et `eq`
- la fonction `Hashset.add` est paramétrée par `'a`, `hash` et `eq`

C'est lourd, et c'est dangereux car on peut appeler `add` et `find` à différents moments avec différentes fonctions `hash` et `eq`, d'où violation de l'invariant.

Solution 1 : foncteur et générativité

Pour paramétrer d'un seul coup vis-à-vis d'un type et deux fonctions, définissons « un type muni de fonctions d'équivalence et de hachage » :

```
module type HashedType = sig
  type t
  val equal: t -> t -> bool
  val hash: t -> int
end
```

Solution 1 : foncteur et générativité

Puis, définissons un foncteur dont la déclaration (dans `hashset.mli`) sera :

```
module Make (H : HashedType) : sig
           (* H.t is the type of elements *)
  type t   (*   t is the type of sets     *)
  val create: int -> t
  val add: t -> H.t -> unit
  val find: t -> H.t -> H.t option
end
```

Les fonctions `equal` et `hash` sont fixées lors de l'appel à `Make`.

Et deux appels à `Make` produisent deux types distincts.

Donc, plus de danger. Problème résolu !

Solution 2 : attacher les fonctions à la table

On peut stocker les fonctions `hash` et `eq` dans la table lors de sa création.

Dans `hashset.mli`, on aura alors :

```
type 'a t
val create: ('a -> int) -> ('a -> 'a -> bool) -> int -> 'a t
val add: 'a t -> 'a -> unit
val find: 'a t -> 'a -> 'a option
```

Ainsi, `hash` et `eq` ne sont plus arguments de `add` et `find`.

Problème résolu aussi !

Solution 2 : attacher les fonctions à la table

Cette deuxième solution est employée dans la bibliothèque de Java.

```
public class TreeSet<E> {
  // Ask for a comparator when a new set is created:
  public TreeSet (Comparator<E> c) { ... }
  // No need to ask for a comparator when using the set:
  public boolean add (E e) { ... }
  public boolean contains (E e) { ... }
}
```

Comparaison des solutions

La première approche fixe `equal` et `hash` quand on applique le foncteur, c'est-à-dire au moment où on crée un module « ensembles ».

- On obtient un nouveau type abstrait des ensembles.
- On peut ensuite créer de nombreux ensembles de ce type,
- et leur appliquer des opérations, y compris l'opération binaire `union`.

La seconde fixe `equal` et `hash` quand on crée un nouvel ensemble vide.

- Tous les ensembles ont le même type.
- Mais deux ensembles peuvent avoir des `equal` et `hash` différents !
- Cette approche est donc inapplicable si on a une opération binaire.

Plusieurs styles

Dans le cas des « hash sets », nous avons vu deux façons raisonnables de présenter le code :

- soit un module paramétré par un module,
- soit un type et des fonctions paramétrés par un type et des fonctions.

Paramétrer en masse ou bien individuellement ? Pas toujours évident.

Plusieurs styles

De même, un « type ordonné » peut être représenté soit par un module :

```
module type OrderedType = sig
  type t
  val compare: t -> t -> int
end
```

soit simplement par une fonction :

```
type 'a comparator =
  'a -> 'a -> int
```

Plusieurs styles

Au lieu du foncteur `Lexico`, nous aurions pu écrire une fonction :

```
let lexico compare1 compare2 (x1, x2) (y1, y2) =
  let c1 = compare1 x1 y1 in
  if c1 <> 0 then c1 else compare2 x2 y2
```

Elle a le type :

```
val lexico:
  'a comparator -> 'b comparator -> ('a * 'b) comparator
```

Elle a même un type plus général, qu'OCaml peut inférer pour nous :

```
val lexico:
  ('a -> 'b -> int) ->
  ('c -> 'd -> int) ->
  ('a * 'c -> 'b * 'd -> int)
```

Interface de Java = type flèche d'OCaml

J'ai dit (séance 3) qu'une interface de Java :

```
public interface Comparator<T> {
  int compare (T o1, T o2);
}
```

correspond à un type de fonction d'OCaml :

```
type 'a comparator =
  'a -> 'a -> int
```

Tous deux décrivent un service.

Tous deux sont paramétrés ici par le type des éléments à comparer.

Interface de Java = signature d'OCaml ?

L'interface `Comparator<T>` est-elle analogue à la signature `OrderedType` ?

```
module type OrderedType = sig
  type t
  val compare: t -> t -> int
end
```

Pas vraiment.

`OrderedType` exige qu'il existe un type `t` tel que `compare` accepte deux éléments de type `t`, mais ne dit pas quel est ce type `t`.

`Comparator<T>` serait plutôt analogue à `OrderedType with type t = T`.

Interface de Java = signature d'OCaml ?

Quel serait l'analogue en Java de cette signature ?

```
sig
  type element
  type set
  val empty: set
  val add: element -> set -> set
  val mem: element -> set -> bool
  val union: set -> set -> set
end
```

Interface de Java = signature d'OCaml ?

Une interface Java ne peut pas contenir de types.

On peut songer à une interface paramétrée par deux types :

```
public interface SetImpl<E, S> {
  S empty ();
  S add (E e, S s);
  boolean mem (E e, S s);
  S union (S s1, S s2);
}
```

Notez que cette interface ne décrit pas « un ensemble »
mais « une implémentation des ensembles ».

Imiter un foncteur en Java ?

Si on a besoin d'une implémentation des ensembles d'entiers, on écrit que pour tout type `S` et pour tout objet de type `SetImpl<Integer, S>`, ...

Ça ressemble à un foncteur.

Mais si on construit une implémentation des ensembles d'entiers, on veut écrire qu'il existe un type `S` tel que nous proposons un objet de type `SetImpl<Integer, S>`.

On ne peut pas écrire cela en Java.

Je ne vois pas d'analogue Java du foncteur `Set.Make` et de la généralité.

Et ceci ?

Peut-être écririez-vous plutôt cette signature :

```
public interface ISet<E> {
  ISet<E> add (E e);
  boolean mem (E e);
  ISet<E> union (ISet<E> s);
}
```

Elle décrit « un ensemble », pas « une implémentation des ensembles ».

Telle quelle, elle est impossible à implémenter... ! (Pourquoi ?)

Et ceci ?

La méthode `union` sait seulement que son argument `s` a le type `ISet<E>`.

Donc, `s` supporte `add`, `mem`, `union`.

Mais aucun moyen d'accéder à ses éléments !

Pour tenter de résoudre ce problème,

- on peut ajouter à `ISet<E>` une méthode pour itérer sur un ensemble,
- mais même ainsi, `union` ne pourra être implémentée que par insertion itérée, ce qui n'est pas forcément optimal.

On bute donc à nouveau sur l'opération binaire `union`.

Que peut faire une fonction polymorphe ?

Que peut être cette fonction ?

```
val mystery: 'a -> int
```

Son argument est de type inconnu...

Que peut faire une fonction polymorphe ?

Que peut être cette fonction ?

```
val mystery: 'a -> int
```

Son argument est de type inconnu... donc elle ne peut rien en faire !

On sent qu'elle est nécessairement constante. (*)

Elle est insensible à une transformation de son argument :

```
mystery (f x) = mystery x
```

(*) Hypothèse simplificatrice

Une fonction peut avoir un effet de bord :

- ne pas terminer,
- lancer une exception,
- modifier une variable globale, etc.

De plus, certaines opérations primitives permettent d'exploiter un objet sans connaître son type :

- en OCaml, = permet de comparer deux objets ;
- en Java, `instanceof` permet d'inspecter l'étiquette d'un objet.

Ici, pour simplifier, on interdit tout cela.

On considère donc un langage réduit avec types algébriques, fonctions, et polymorphisme, mais rien d'autre.

Que peut faire une fonction polymorphe ?

Que penser de cette autre fonction-mystère ?

```
val mystery: 'a list -> int
```

Les éléments de la liste sont de type inconnu...

Que peut faire une fonction polymorphe ?

Que penser de cette autre fonction-mystère ?

```
val mystery: 'a list -> int
```

Les éléments de la liste sont de type inconnu... donc elle n'y a pas accès.

La fonction est insensible à une transformation de son argument :

```
mystery (map f xs) = mystery xs
```

Cela implique qu'elle n'exploite que la longueur de la liste.

On le voit en choisissant `f = fun x -> ()`.

Que peut faire une fonction polymorphe ?

Que penser de celle-ci ?

```
val mystery: 'a list -> 'a list
```

Que peut faire une fonction polymorphe ?

Que penser de celle-ci ?

```
val mystery: 'a list -> 'a list
```

La longueur du résultat ne dépend que de la longueur de l'argument...

Que peut faire une fonction polymorphe ?

Que penser de celle-ci ?

```
val mystery: 'a list -> 'a list
```

La longueur du résultat ne dépend que de la longueur de l'argument...

Les éléments du résultat proviennent nécessairement de l'argument...

Que peut faire une fonction polymorphe ?

Que penser de celle-ci ?

```
val mystery: 'a list -> 'a list
```

La longueur du résultat ne dépend que de la longueur de l'argument...

Les éléments du résultat proviennent nécessairement de l'argument...

Le choix des paires (i, j) tel que le i-ème élément de l'argument devient le j-ème élément du résultat ne dépend pas des éléments !

La fonction est insensible à une transformation de son argument :

```
mystery (map f xs) = map f (mystery xs)
```

Que peut faire une fonction polymorphe ?

Que penser enfin de celle-ci ?

```
val mystery: ('a * 'a -> bool) * 'a list -> 'a list
```

et de celle-ci ?

```
val mystery: ('a * 'a -> 'a * 'a) * 'a list -> 'a list
```

Que peut faire une fonction polymorphe ?

Que penser enfin de celle-ci ?

```
val mystery: ('a * 'a -> bool) * 'a list -> 'a list
```

et de celle-ci ?

```
val mystery: ('a * 'a -> 'a * 'a) * 'a list -> 'a list
```

D'après leurs types, ce pourraient être des fonctions de tri.

Que peut faire une fonction polymorphe ?

Que penser enfin de celle-ci ?

```
val mystery: ('a * 'a -> bool) * 'a list -> 'a list
```

et de celle-ci ?

```
val mystery: ('a * 'a -> 'a * 'a) * 'a list -> 'a list
```

D'après leurs types, ce pourraient être des fonctions de tri.

Cette fois, les éléments peuvent être inspectés, mais seulement via la fonction de comparaison fournie !

Ces fonctions sont insensibles à une certaine transformation de leur argument... à condition que la fonction de comparaison le soit également.

Que peut faire une fonction polymorphe ?

Plus précisément, pour la première de ces deux fonctions :

```
val mystery: ('a * 'a -> bool) * 'a list -> 'a list
```

Si cette propriété est satisfaite :

```
c2 (f x) (f y) = c1 x y
```

alors on aura la propriété :

```
mystery c2 (map f xs) = map f (mystery c1 xs)
```

`f` a un type de la forme `t -> u` où `t` et `u` ne sont pas forcément les mêmes.

`f` peut jeter de l'information, mais pas trop, pour que la comparaison reste possible.

Que peut faire une fonction polymorphe ?

Et pour la seconde fonction :

```
val mystery: ('a * 'a -> 'a * 'a) * 'a list -> 'a list
```

Si cette propriété est satisfaite :

```
swap2 (f x, f y) = map_pair f swap1 (x, y)
```

où `map_pair f (x, y) = (f x, f y)`

alors on aura la propriété :

```
mystery swap2 (map f xs) = map f (mystery swap1 xs)
```

Où en sommes-nous ?

Je prétends que

- si on connaît le type d'une fonction (polymorphe),
- (mais pas son code !)
- alors on peut affirmer qu'elle est insensible à une certaine transformation de ses arguments.

Wadler (1989) appelle cela un théorème gratuit.

Où en sommes-nous ?

Mais...

- à quoi ça sert ?
- comment le démontre-t-on ?

Voyons cela très brièvement...

À quoi ça sert ?

Revenons à cette fonction-mystère, qui pourrait être un tri :

```
val mystery: ('a * 'a -> 'a * 'a) * 'a list -> 'a list
```

Du « théorème gratuit », on déduit que si elle trie correctement toute liste de Booléens, alors elle trie correctement toute liste d'éléments de type `t`.

- La preuve (omise) demande un bon choix de `f : t -> bool`.

C'est le principe 0-1 de Knuth (1973).

Intéressant pour limiter les tests à effectuer !

À quoi ça sert ?

De façon analogue, pour celle-ci :

```
val mystery: ('a * 'a -> bool) * 'a list -> 'a list
```

Du « théorème gratuit », on déduit que si elle trie correctement toute liste d'entiers, alors elle trie correctement toute liste d'éléments de type `t`.

- La preuve (omise) demande un bon choix de `f : t -> int`.

Intéressant pour limiter les tests à effectuer !

Comment le démontre-t-on ?

Traditionnellement, on interprète un type T comme un ensemble de valeurs, et on démontre un théorème qui affirme que les types décrivent correctement les valeurs :

> *Si (le compilateur a vérifié que) l'expression e a le type T, alors (à l'exécution) l'expression e produit une valeur v telle que* $v \in T$.

Reynolds (1983) généralise cela et interprète un type T comme une relation entre deux ensembles T_1 et T_2. Il démontre alors ceci :

> *Si (le compilateur a vérifié que) l'expression e a le type T, alors (à l'exécution) l'expression e produit une valeur v telle que* $v\ T\ v$.

Ce résultat est appelé théorème de paramétricité. (Preuve omise.)

Comment le démontre-t-on ?

Pour comprendre l'énoncé de Reynolds, il faut lire un type T comme une relation entre deux ensembles T_1 et T_2.

Si $x_1 \in T_1$ et $x_2 \in T_2$, la relation $x_1\ T\ x_2$ signifie intuitivement que x_1 et x_2 sont indistinguables.

Reynolds définit ainsi la façon dont on lit un type comme une relation :

$$
\begin{array}{rcll}
b_1 & \mathit{Bool} & b_2 & \text{ssi}\quad b_1 = b_2 \\
(a_1, b_1) & (A \times B) & (a_2, b_2) & \text{ssi}\quad a_1\ A\ a_2 \wedge b_1\ B\ b_2 \\
[] & (\mathit{List}\ A) & [] & \\
a_1 :: as_1 & (\mathit{List}\ A) & a_2 :: as_2 & \text{ssi}\quad a_1\ A\ a_2 \wedge as_1\ (\mathit{List}\ A)\ as_2 \\
f_1 & (A \to B) & f_2 & \text{ssi}\quad \forall a_1 a_2 \quad a_1\ A\ a_2 \Rightarrow f_1(a_1)\ B\ f_2(a_2) \\
t_1 & (\forall A.T) & t_2 & \text{ssi}\quad \forall A_1, A_2, A : \mathit{Rel}(A_1, A_2) \quad t_1\ T\ t_2
\end{array}
$$

Exemple

Voyons ce qu'affirme le théorème de Reynolds pour l'une de nos fonctions *mystery*.

Elle est reliée à elle-même par son type-interprété-comme-relation :

$$\mathit{mystery} \quad (\forall A.((A \times A \to \mathit{Bool}) \times \mathit{List}\ A) \to \mathit{List}\ A) \quad \mathit{mystery}$$

Exemple

Commençons à déplier la définition :

$$\forall A_1, A_2, A : Rel(A_1, A_2)$$
$$mystery \quad (((A \times A \to Bool) \times List\ A) \to List\ A) \quad mystery$$

Ici A_1, A_2 sont des ensembles est A est une relation entre eux.

Exemple

Déplions encore :

$$\begin{array}{l}
\forall A_1, A_2, A : \mathit{Rel}(A_1, A_2) \\
\forall (\prec_1), (\prec_2), \mathit{as}_1, \mathit{as}_2 \\
(\prec_1)\ (A \times A \to \mathit{Bool})\ (\prec_2) \quad \wedge \\
\mathit{as}_1\ (\mathit{List}\ A)\ \mathit{as}_2 \quad \Rightarrow \\
\mathit{mystery}(\prec_1, \mathit{as}_1) \quad (\mathit{List}\ A) \quad \mathit{mystery}(\prec_2, \mathit{as}_2)
\end{array}$$

Ici $\prec_i$ est une fonction dans $A_i \times A_i \to \mathit{Bool}$,
et as_i est une liste dans $\mathit{List}\ A_i$.

Exemple

Déplions toujours :

$$\begin{array}{l}
\forall A_1, A_2, A : \mathit{Rel}(A_1, A_2) \\
\forall (\prec_1), (\prec_2), \mathit{as}_1, \mathit{as}_2 \\
(\forall x_1, x_2, y_1, y_2 \quad x_1 \ A \ x_2 \wedge y_1 \ A \ y_2 \Rightarrow (x_1 \prec_1 y_1) = (x_2 \prec_2 y_2)) \quad \wedge \\
\mathit{as}_1 \ (\mathit{List}\ A) \ \mathit{as}_2 \quad \Rightarrow \\
\mathit{mystery}(\prec_1, \mathit{as}_1) \quad (\mathit{List}\ A) \quad \mathit{mystery}(\prec_2, \mathit{as}_2)
\end{array}$$

Ici x_i et y_i sont des éléments de A_i.

Exemple

Restreignons cette affirmation au cas où la relation A entre A_1 et A_2 est en fait une fonction f dans $A_1 \to A_2$.

La relation $x_1\ A\ x_2$ devient $f(x_1) = x_2$.

La relation $as_1\ (List\ A)\ as_2$ devient $map\ f\ as_1 = as_2$.

On trouve alors :

$$\begin{array}{l}
\forall A_1, A_2, f : A_1 \to A_2 \\
\forall (\prec_1), (\prec_2), as_1 \\
(\forall x_1, y_1 \quad (x_1 \prec_1 y_1) = (f(x_1) \prec_2 f(y_1))) \quad \Rightarrow \\
map\ f\ (mystery(\prec_1, as_1)) = mystery(\prec_2, map\ f\ as_1)
\end{array}$$

On retrouve le « théorème gratuit » énoncé sans preuve plus tôt.

Exemple

On peut donc affirmer, sans connaître le code de la fonction *mystery* :

> *Pour tous types* A_1, A_2 *et pour toute transformation* $f : A_1 \to A_2$,
> *pour toutes fonctions de comparaison* $(\prec_1), (\prec_2)$,
> *si* f *préserve les comparaisons, alors*
> *appliquer mystery avec* $\prec_1$ *puis appliquer* f
> *est équivalent à*
> *appliquer* f *puis appliquer mystery avec* $\prec_2$.

Ce théorème résume le fait que *mystery* fonctionne de la même manière quel que soit le type `'a`.

À retenir

À propos des modules et foncteurs :

- on peut voir un composant comme un groupe de types et de valeurs ;
- paramétrer un composant par un composant conduit à une façon naturelle d'assembler des composants.

À propos de la paramétricité :

- du type d'une fonction polymorphe, on peut déduire des informations sur son comportement.

En TD aujourd'hui :

- test d'un objet décrit uniquement par une interface de Java.