TYPES ABSTRAITS : PLUSIEURS EXEMPLES

Note : les exemples sont écrits en pseudo-code semblable à du CAML, mais ce n'est pas du CAML.

1 Type abstrait Pile

On donne ici deux représentations des piles en version mutable (programmation impérative) ou permanente (programmation fonctionnelle).

1.1 Type mutable

Soit E un ensemble représenté par un type `e`. On définit le type « pile d'éléments de E » comme le type abstrait **mutable** suivant (noté `e pile`) défini par ses fonctions et ses relations :

fonctions :

- `new ()` de type `unit -> e pile` : création d'une pile vide ;
- `push x p` de type `e -> e pile -> unit` : ajout de l'élément `x` au sommet de la pile `p` (pas de création d'une nouvelle pile, mais plutôt modification de la pile directement en place) ;
- `pop p` de type `e pile -> e` : récupération du sommet de la pile **et** modification de la pile ;
- `empty p` de type `e pile -> bool` : vacuité de la pile ou non.

relations (`x` est un élément quelconque, `p` est une pile) :

- `empty (new ())` est toujours vrai
- `(push x p; empty p)` est toujours faux
- `pop (new ())` n'est pas défini (erreur)
- `(push x p; pop p)` est toujours égal à `x` et ne modifie pas l'état de la pile
- `(x = pop p; push x p)` ne modifie pas l'état de la pile
- etc

1.2 Type permanent

Soit E un ensemble représenté par un type `e`. On définit le type « pile d'éléments de E » comme le type abstrait **permanent** suivant (noté `e pile`) défini par ses fonctions et ses relations :

fonctions :

- `new ()` de type `unit -> e pile` : création d'une pile vide ;
- `push x p` de type `e -> e pile -> e pile` : création d'une nouvelle pile avec ajout de l'élément `x` au sommet de la pile `p` ;
- `pop p` de type `e pile -> e * e pile` : récupération du sommet de la pile **et** pile sans son sommet ;
- `empty p` de type `e pile -> bool` : vacuité de la pile ou non.

relations (`x` est un élément quelconque, `p` est une pile) :

- `empty (new ())` est toujours vrai
- `(let q = push x p in empty q)` est toujours faux
- `pop (new ())` n'est pas défini (erreur)
- `(let q = push x p in pop q)` est toujours égal à `x, p`
- `(let x, q = pop p in push x q)` calcule une pile identique à `p`
- etc

1.3 En pratique

On ajoute que ces opérations se font en complexité constante.

Autrement dit, une pile est une structure mutable LIFO (Last In, First Out) : le dernier élément rentré est le premier à sortir.

Exemple d'utilisation : évaluation des expressions algébriques postfixes avec une pile, évaluation des expressions algébriques infixes (= classiques) avec deux piles, compilateurs, récursivités dans un langage non récursif, équilibrage de parenthèses, etc

CAML offre le type pile dans son module `stack`, avec d'autres fonctions utiles, mais pas la fonction `empty` : rien de grave, on peut se débrouiller autrement.

2 Type abstrait File

2.1 Type mutable

Soit E un ensemble représenté par un type `e`. On définit le type « file d'éléments de E » comme le type abstrait **mutable** suivant (noté `e file`) défini par ses fonctions et ses relations :

fonctions :

- `new ()` de type `unit -> e file` : création d'une file vide ;
- `add x f` de type `e -> e file -> unit` : ajout de l'élément `x` à l'entrée de la file `f` (modification de la file directement en place) ;
- `take f` de type `e file -> e` : récupération d'un objet à la sortie de la file **et** modification de la file ;
- `empty f` de type `e file -> bool` : vacuité de la file ou non.

relations (`x` est un élément quelconque, `f` est une file) :

- `empty (new ())` est toujours vrai
- `(take x f; empty f)` est toujours faux
- `take (new ())` n'est pas défini (erreur)
- si `f` est une file vide, `(add x f; take f)` est toujours égal à `x` et la file reste vide
- si `f` est une file non vide, `(add x f; take f)` est toujours égal à `(y = take f; add x f; y)` et l'état de la file est identique dans les deux cas
- etc

2.2 Type permanent

Soit E un ensemble représenté par un type `e`. On définit le type « file d'éléments de E » comme le type abstrait **permanent** suivant (noté `e file`) défini par ses fonctions et ses relations :

fonctions :

- `new ()` de type `unit -> e file` : création d'une file vide ;
- `add x f` de type `e -> e file -> e file` : création d'une nouvelle file par ajout de l'élément `x` à l'entrée de la file `f` ;
- `take f` de type `e file -> e * e file` : récupération d'un objet à la sortie de la file **et** nouvelle file privée de l'objet récupéré ;
- `empty f` de type `e file -> bool` : vacuité de la file ou non.

relations (`x` est un élément quelconque, `f` est une file) :

- `empty (new ())` est toujours vrai
- `(let g = add x f in empty g)` est toujours faux
- `take (new ())` n'est pas défini (erreur)
- si `f` est une file vide, `(let g = add x f in take g)` est toujours égal à `(x, f)`
- si `f` est une file non vide, `(let g = add x f in take f)` est toujours égal à `(let y, g = take f in add x g; (y, g))`
- etc

2.3 En pratique

On ajoute que ces opérations se font en complexité constante en moyenne (complexité amortie).

Autrement dit, une file est une structure mutable FIFO (First In, First Out) : le premier élément rentré est le premier à sortir.

Exemple d'utilisation : files d'attentes en gestion de ressources, parcours en largeur d'un arbre, gestion des événements en programmation événementielle, etc.

CAML offre le type file dans son module `queue`, avec d'autres fonctions utiles, mais pas la fonction `empty` : rien de grave, on peut se débrouiller autrement.

3 Type abstrait File de priorité

3.1 Type mutable

Soit E un ensemble *ordonné* représenté par un type `e`. On définit le type « file de priorité d'éléments de E » comme le type abstrait **mutable** suivant (noté `e prior`) défini par ses fonctions et ses relations :

fonctions :

- `new ()` de type `unit -> e prior` : création d'une file vide ;
- `add x f` de type `e -> e prior -> unit` : ajout de l'élément `x` dans la file de priorité `f` (modification de la file de priorité directement en place) ;
- `peek f` de type `e prior -> e` : récupération de l'objet minimal de la file de priorité **sans** modification de la file ;
- `take f` de type `e prior -> e` : récupération de l'objet minimal de la file de priorité **et** modification de la file ;
- `empty f` de type `e prior -> bool` : vacuité de la file ou non ;
- `length f` de type `e prior -> int` : nombre d'objets dans la file.

relations (`x` est un élément quelconque, `f` est une file de priorité) :

- `empty (new ())` est toujours vrai
- `(add x p; empty p)` est toujours faux
- `take (new ())` et `peek (new ())` ne sont pas définis (erreur)
- si `f` est une file de priorité vide, `(add x f; take f)` est toujours égal à `x` et la file reste vide
  `(add x f; peek f)` est toujours égal à `x` et la file est non vide ;
- si `f` est une file de priorité non vide, soit `y = peek f`, alors
  `add x f; take f` est égal à `x` si `x <= y` et l'état de la file est inchangé
  `add x f; take f` est égal à `y` si `y < x` et l'état de la file est modifié
- etc

3.2 Type permanent

Soit E un ensemble *ordonné* représenté par un type `e`. On définit le type « file de priorité d'éléments de E » comme le type abstrait **permanent** suivant (noté `e prior`) défini par ses fonctions et ses relations :

fonctions :

- `new ()` de type `unit -> e prior` : création d'une file vide ;
- `add x f` de type `e -> e prior -> e prior` : création d'une nouvelle file par ajout de l'élément `x` dans la file de priorité `f` ;
- `peek f` de type `e prior -> e` : récupération de l'objet minimal de la file de priorité **sans** modification de la file ;
- `take f` de type `e prior -> e * e prior` : récupération de l'objet minimal de la file de priorité **et** nouvelle file privée de son élément minimal ;
- `empty f` de type `e prior -> bool` : vacuité de la file ou non ;
- `length f` de type `e prior -> int` : nombre d'objets dans la file.

relations (`x` est un élément quelconque, `f` est une file de priorité) :

- `empty (new ())` est toujours vrai
- `(let q = add x p in empty p)` est toujours faux
- `take (new ())` et `peek (new ())` ne sont pas définis (erreur)
- si `f` est une file de priorité vide, `(let g = add x f in take f)` est toujours égal à `(x, f)`
  `(let g = add x f in peek f)` est toujours égal à `x` ;
- si `f` est une file de priorité non vide, soit `let y = peek f`, alors
  `let h = add x f in take h` est égal à `x, f` si `x <= y` et l'état de la file est inchangé
  `let h = add x f in take f` est égal à `y, f'` si `y < x`
- etc

3.3 En pratique

On ajoute que ces opérations se font si possible en complexité $O(\log n)$, où n est le nombre d'éléments de la file de priorité.

Exemple d'utilisation : tri par tas, alg. de Dijkstra.

CAML n'offre pas de module de ce type, c'est à l'utilisateur de choisir son implémentation.

4 Type abstrait Dictionnaire

4.1 Type mutable

Soit K, E deux ensembles représentés par deux types k et e. On définit le type « dictionnaire à clefs dans K et valeurs dans E » comme le type abstrait **mutable** suivant (noté `(k,e) dict`) défini par ses fonctions et ses relations :

fonctions :

- `new ()` de type `unit -> (k,e) dict` : création d'un dictionnaire vide ;
- `add c x d` de type `k-> e -> (k,e) dict -> unit` : ajout de l'élément `x` associé à la clef `c` dans le dictionnaire `d` (modification du dictionnaire directement en place) ;
- `find c d` de type `k -> (k,e) dict -> e` : récupération de l'objet du dictionnaire associé à la clef `c` **sans** modification du dictionnaire, ou erreur ;
- `remove c d` de type `k -> (k,e) dict -> unit` : suppression de l'objet du dictionnaire associé à la clef `c` (donc modification du dictionnaire) ou erreur ;
- `empty d` de type `(k,e) dict -> bool` : vacuité du dictionnaire ou non ;
- `length d` de type `(k,e) dict -> int` : nombre d'objets dans le dictionnaire.

relations (`x` est un élément quelconque, `c` une clef, `d` est un dictionnaire) :

- `empty (new ())` est toujours vrai
- `(add c x d; empty d)` est toujours faux
- `find c (new ())` et `remove c (new ())` ne sont pas définis (erreur)
- `add c x d; find c d` est toujours égal à `x` et l'état du dictionnaire est modifié
- `remove c x d; find c d` est toujours une erreur
- `add c x d; remove c d` ne modifie pas le dictionnaire
- etc

4.2 Type permanent

Soit K, E deux ensembles représentés par deux types k et e. On définit le type « dictionnaire à clefs dans K et valeurs dans E » comme le type abstrait **permanent** suivant (noté `(k,e) dict`) défini par ses fonctions et ses relations :

fonctions :

- `new ()` de type `unit -> (k,e) dict` : création d'un dictionnaire vide ;
- `add c x d` de type `k-> e -> (k,e) dict -> (k,e) dict` : création d'un nouveau dictionnaire par ajout de l'élément `x` associé à la clef `c` dans le dictionnaire `d` ;
- `find c d` de type `k -> (k,e) dict -> e` : récupération de l'objet du dictionnaire associé à la clef `c` **sans** modification du dictionnaire, ou erreur ;
- `remove c d` de type `k -> (k,e) dict -> (k,e) dict` : création d'un nouveau dictionnaire par suppression de l'objet du dictionnaire associé à la clef `c` ou erreur ;
- `empty d` de type `(k,e) dict -> bool` : vacuité du dictionnaire ou non ;
- `length d` de type `(k,e) dict -> int` : nombre d'objets dans le dictionnaire.

relations (`x` est un élément quelconque, `c` une clef, `d` est un dictionnaire) :

- `empty (new ())` est toujours vrai
- `(let d' = add c x d in empty d)` est toujours faux
- `find c (new ())` et `remove c (new ())` ne sont pas définis (erreur)
- `let d' = add c x d in find c d'` est toujours égal à `x`
- `let d' = remove c x d in find c d` est toujours une erreur
- `let d' = add c x d in remove c d'` est toujours égal à `d`
- etc

4.3 En pratique

Le type dictionnaire est une généralisation du type tableau : un tableau est un dictionnaire dont les clefs sont des entiers consécutifs, mais dont la taille est immuable.

Exemple d'utilisation : mémoïsation, etc

CAML offre le type dictionnaire dans son module `hashtbl`, avec d'autres fonctions utiles, mais pas la fonction `empty`, ni la fonction `length` : rien de grave, on peut se débrouiller autrement.

La complexité n'est pas toujours spécifiée, mais on attend souvent au plus $O(n)$ où n est le nombre d'associations du dictionnaire. On peut même obtenir $O(\log n)$.