Types sommes et enregistrements

Jean-Baptiste Rouquier

1 types sommes

1.1 Présentation

Un type somme est une disjonction entre plusieurs types. On peut n'avoir que des constructeurs ***constants*** comme dans

```
type couleur = Pique | Coeur | Carreau | Trefle
```

ou bien des ***constructeurs*** prenant un argument comme dans

```
type nombre = Entier of int | Flottant of float
```

Question 1.1. Écrire l'une des deux fonctions suivantes (au choix) :

version courte Écrire la fonction `ajoute : nombre -> nombre -> nombre` qui renvoit un `Entier` si les deux arguments sont des `Entier`, un `Flottant` si les deux arguments sont des `Flottant`, une erreur si les types sont incompatibles.

plus long Écrire la fonction `divise : nombre -> nombre -> nombre` qui renvoit un `Entier` si les deux arguments sont des `Entier` et que la division tombe juste, un `Flottant` sinon.

La tester.

1.2 Réinventer la roue

Les listes pourraient être écrites

```
type 'a liste = Nil | Cons of 'a * 'a liste
```

Question 1.2. Réimplémenter `List.tl` (renvoie la liste sans son premier élément), `List.rev` (retourne la liste) et `List.map` (applique une fonction à chacun des éléments d'une liste et renvoie la liste des résultats) sur ce type. Les tester.

Remarque. Dans la suite on utilisera les listes natives de Caml, pas ce nouveau type.

2 types enregistrement

2.1 explications

Un type ***enregistrement*** consiste en plusieurs ***champs*** ayant chacun leur type. Exemple pour un élément d'un menu de restaurant :

```
type categorie = Entree | Principal | Dessert
type plat = {nom:string; prix:int; vegetarien:bool; categorie:categorie}
```

On définit une variable de ce type en remplissant tous les champs (pas forcément dans l'ordre) :

```
let foo = {nom="salade"; vegetarien=true; prix=8; categorie=Entree}
```

On lit la valeur d'un champ par `foo.nom`. On peut faire du filtrage (pas nécessairement sur tous les champs) soit comme sur les sommes :

```
let is_entree_vegetarienne = function
  | {categorie=Entree; vegetarien=true} -> true
  | _ -> false
```

soit directement sur l'argument donné :

```
let pas_cher {prix=p} =
  p < 10
```

Enfin une syntaxe pratique : reprendre tous les champs d'une variable en en modifiant quelques uns :

```
let ajoute_pourboire plat = {plat with prix = plat.prix + 3}
```

Remarque. Caml donne les types

```
val pas_cher : plat -> bool = <fun>
val is_entree_vegetarienne : plat -> bool = <fun>
val ajoute_pourboire : plat -> plat = <fun>
```

Remarque. Tant que l'on ne connaît pas les modules (et l'on n'en a pas besoin en prépa), on ne peut pas avoir deux fois le même nom pour un champ dans des types distincts.

2.2 applications

2.2.1 mise en bouche

On pose `type fraction = {numerateur:int; denominateur:int}`.

Question 2.1. Écrire la fonction `ajoute: fraction -> fraction -> fraction` (on ne se préoccupera pas de garder des fractions irréductibles).

2.2.2 champs modifiables

Les références sont implémentées ainsi en OCaml :

```
type 'a reference = {mutable contenu : 'a}
let get reference = reference.contenu
let set reference valeur = reference.contenu <- valeur
```

Le mot clef `mutable` signifie que ce champ peut être modifié, par la syntaxe `<-`.

2.2.3 la file de la cantine

Rappel : une ***pile*** (LIFO) permet d'ajouter un élément au début et de retirer l'élément du début. Une ***file*** (FIFO) permet d'ajouter un élément à la fin et de retirer l'élément du début.

On va implémenter une version rapide d'un modèle plus réaliste avec insertion et suppression aux deux bouts. L'idée est de garder deux listes, l'une contenant le début de la file, l'autre contenant la fin. La première liste, `debut`, est à l'endroit : son premier élément est le premier élément de la file. L'autre, `fin`, est à l'envers : son premier élément est le dernier de la file.

Ajouter un élément au début où à la fin se fait sur la file corespondante. Retirer un élément au début ou à la fin se fait sur la file correspondante si possible ; sinon, par exemple si `debut` est vide, on la remplace par `fin` retournée, et la file de la fin devient vide.

Question 2.2. Choisir entre version fonctionnelle et version impérative et implémenter les fonctions suivantes :

- `ajoute` ajoute un élément à la fin ;
- `prend` prend l'élément du début ;
- `gruge` ajoute un élément au début ;
- `retire` retire le dernier élément.

version fonctionnelle

```
type 'a file = {debut:'a list; fin: 'a list}
val empty : 'a file
val ajoute : 'a file -> 'a -> 'a file
val prend : 'a file -> 'a * 'a file
val gruge : 'a file -> 'a -> 'a file
val retire : 'a file -> 'a * 'a file
```

version impérative

```
type 'a file = {mutable debut:'a list; mutable fin: 'a list}
val create : unit -> 'a file
val ajoute : 'a file -> 'a -> unit
val prend : 'a file -> 'a
val gruge : 'a file -> 'a -> unit
val retire : 'a file -> 'a
```

Question 2.3. Tester ces fonctions.

Question 2.4 (bonus). Trouver une suite d'opérations où le coût amorti (ie le coût total divisé par le nombre d'opérations) n'est pas constant mais $\Theta(n)$ où n est le nombre d'éléments de la file.

Pour pallier ce problème, on propose de ne pas retourner toute la liste quand on veut retirer un élément du côté où la liste est vide. Réécrire les questions précédentes pour ne retourner que la moitié de la liste. Toutes les opérations se font alors en coût amorti constant.

Remarque. Des listes doublement chaînées ne permettent pas d'écrire une version fonctionnelle. Kaplan, Okasaki et Tarjan ont proposé une structure de données fonctionnelle implémentant les files avec insertion et suppression au deux bouts, ainsi que la concaténation de deux files, le tout en temps (amorti) constant.