TD #6 — Types somme, filtrage… (avec solutions)

Exercice 1. *Inférence de types*

Donnez les **types OCaml** de chacune des expressions ci-dessous :

1. `[1; 2; 3]`

   ```
   int list
   ```

2. `[1, 2, 3]`

   ```
   (int * int * int) list
   ```

3. `[[]; []]`

   ```
   'a list list
   ```

4. `None`

   ```
   'a option
   ```

5. `Some (Some None)`

   ```
   'a option option option
   ```

6. `fun x d -> match x with Some v -> v | None -> d`

   ```
   'a option -> 'a -> 'a
   ```

7. `function x::_ -> Some x | _ -> None`

   ```
   'a list -> 'a option
   ```

8. `function (_,x)::_ -> Some x | _ -> None`

   ```
   ('a * 'b) list -> 'b option
   ```

9. `fun () -> 0`

   ```
   unit -> int
   ```

10. `fun 3 -> true`

    ```
    int -> bool
    ```

La fonction définie par cette dernière expression est problématique…

11. Donnez deux exemples d'appel de cette fonction qui illustrent son comportement et ce problème.

    On a par exemple :
    ```
    (fun 3 -> true) 3 (* true *)
    (fun 3 -> true) 4 (* Exception: Match_failure *)
    ```

12. Pourquoi la fonction de l'avant-dernière expression n'est-elle *pas* problématique ?

    Le type `unit` possède exactement un habitant ; il n'y a donc qu'un seul cas à considérer dans un filtrage, qui est forcément exhaustif.

Exercice 2. *Définition de type*

1. Définissez un type somme permettant de représenter ou bien un `int` ou bien un `bool`

On propose :
```
type ib = Int of int | Bool of bool
```

2. Donnez un exemple de construction d'une valeur de ce type pour chacun des cas.

Par exemple :
```
Int 12
Bool false
```

3. Écrivez une fonction OCaml `ibadd : ib -> ib -> int` qui additionne deux valeurs de type `ib` en traitant `false` comme valant `0` et `true` comme `1`

On propose :
```
let ibunpck = function Int x -> x | Bool x -> if x then 1 else 0
let ibadd x y = (ibunpck x) + (ibunpck y)
```

Exercice 3. *Listes d'association*

On définit informellement une *liste d'association* comme une liste OCaml de type `('a * 'b) list`, où chaque couple `(k, v:'a * 'b)` présent dans la liste « associe » la *valeur* `v` à la *clef* `k`.

1. Écrivez une fonction OCaml :
   `find : 'a -> ('a * 'b) list -> 'b option`
   telle que pour `(x: ('a * 'b) list)`, et `(k : 'a)`, `find k x` s'évalue en `Some v` avec `v` la première (en partant de la tête) valeur telle que `k, v` apparaisse dans `x` si une telle valeur existe, et `None` sinon.

On propose :
```
let rec find k = function
    | [] -> None
    | (k', v)::xs -> if k' = k then Some v else find k xs
```

2. Illustrez l'utilisation de `find` avec trois exemples d'appel (et d'évaluation associée) bien choisis.

Par exemple :
```
find 3 [(3,4)] (* Some 4 *)
find 3 [(3,5); (3,4)] (* Some 5 *)
find 4 [(3,4)] (* None *)
```

3. Déterminez le coût temporel pire cas de votre fonction `find`, et exprimez le de façon asymptotique en fonction d'un paramètre pertinent.

Le pire cas correspond à un premier argument `k` qui n'apparaît comme aucune clef dans `x` ; dans ce cas, `find` parcourt l'intégralité de la liste avant de s'évaluer à `None`.

Soit n la longueur de la liste `x`, on va prouver par récurrence :
$\mathcal{P}(n)$: un appel à `find _ x` occasionne au plus n appels récursifs (à `find`).
Initialisation ($\mathcal{P}(0)$) : si $n = 0$, `x` ne peut être que la liste vide `[]`. Dans ce cas (lignes 1 & 2) `find` n'émet aucun appel récursif et la propriété est vraie.
Hérédité ($\mathcal{P}(n) \to \mathcal{P}(n+1)$) : on considère un appel à `find` avec un argument `x` de longueur $n+1 > 0$. Par le code de la fonction (lignes 1 & 3), `find` termine (et n'émet donc pas d'appel récursif) ou émet un appel récursif avec comme argument une liste de longueur n. Par hypothèse de récurrence, cet appel à `find` émet au plus n appels récursifs à `find`, et l'appel initial en émet donc au plus $n+1$. La propriété est vraie.
Par le principe de récurrence, $\mathcal{P}(n)$ est vraie pour tout $n \in \mathbb{N}$.

On conclut l'analyse de coût en constatant qu'en dehors d'un éventuel appel récursif, les opérations « marginalement » effectuées par `find` (un filtrage, et éventuellement une extraction de tête de liste et de couple, une comparaison, l'utilisation du constructeur `Some`) sont toutes de coût constant ; le coût pire cas de `find _ x` est donc un $O(n)$, avec n la longueur de `x`.
N.B. : En réalité, le coût d'une comparaison avec `(=)` n'est pas forcément constant (pourquoi ?), mais l'on ignorera ici cette subtilité.

4. Écrivez une fonction OCaml :

`add : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list`
telle que pour `(x: ('a * 'b) list)`, `(k: 'a)`, `(v: 'b)`, `add k v x` s'évalue en une liste identique à `x` à laquelle on a ajouté l'association `(k, v)` *en tête*.

La réponse est plus simple que la question :
```
let add k v x = (k, v)::x
```

5. Écrivez une fonction OCaml :
`del : 'a -> ('a * 'b) list -> ('a * 'b) list`
telle que pour `(x: ('a * 'b) list)`, `(k: 'a)`, `del k x` s'évalue en une liste identique à `x` si aucune association faisant intervenir la clef `k` n'y apparaît, et sinon identique à `x` dont on a « supprimé » la première de ces associations (en partant de la tête).

On propose :
```
let rec del k = function
    | [] -> []
    | (k', v)::xs -> if k' = k then xs else (k', v)::(del k xs)
```

Exercice 4. *Introduction aux exceptions*

OCaml possède un type `exn` de *valeurs exceptionnelles*, ou *exceptions*. Une exception peut être *levée* en lui appliquant la fonction `raise : exn -> 'a`. Le type de `raise` indique *que son évaluation ne terminera jamais* (nous reviendrons là-dessus en cours) : lorsqu'une exception est levée, le flot d'exécution normal du programme est interrompu jusqu'à ce que le programme tout entier termine brutalement, ou que l'exception soit *rattrapée*. Une exception se rattrape avec un filtrage de motif adapté, habituellement en utilisant la construction :
`try en with <Ex1> -> ex1 | <Ex2> -> ex2 ...`
qui s'évalue en `en` si aucune exception ne s'échappe lors de son évaluation, et en `ex1` si une exception correspondant au motif `<Ex1>` est levée (et pas déjà rattrapée) lors de son évaluation, et en `ex2` si une exception correspondant au motif `<Ex2>` (mais pas au motif `<Ex1>`) est…

Soit l'exception prédéfinie `exception Exit` :

1. En quoi s'évalue `try raise Exit with _ -> true` ?

```
true
```

2. En quoi s'évalue `try false with _ -> true` ?

```
false
```

3. Pourquoi l'expression `try false with _ -> None` n'est-elle pas typable ?

Il ne serait pas illogique de vouloir que cette expression s'évalue toujours à `false` et donc qu'elle soit typable de type `bool`. Cependant, ceci implique de considérer ce qu'il se passe à l'exécution (de façon dynamique), et le typage en OCaml est statique. Dans ce cas, on a nécessairement la contrainte que *toutes* les évaluations imaginables de l'expression aient le même type (même si cela pousse à imaginer des choses qui n'arriveront jamais…). Cette contrainte n'est pas satisfaite ici, puisque `(false: bool)` et `(None: 'a option)` ne sont pas de même type.

Soit l'exception prédéfinie `exception Failure of string` dont le constructeur (unaire) s'appelle avec un argument `string` (« chaîne de caractères » ; une `string` s'écrit entre deux « " ») :

4. Écrivez une fonction `head : 'a list -> 'a` qui s'évalue en la tête de son argument si elle en possède une, et lève une exception `Failure` sinon.

On propose :
```
let head = function x::_ -> x | _ -> raise (Failure "empty list")
```

5. Illustrez son comportement avec trois exemples d'appels bien choisis.

Par exemple :
```
head []   (* Exception: Failure "empty list" *)
head [12] (* 12 *)
try head [12] with Failure _ -> 0 (* 12 *)
```

6. Quelle alternative à une exception pourrait-on employer pour une telle fonction `head`, en changeant un peu d'approche (et son type) ?

On pourrait utiliser une approche où le résultat serait de type option :

```
let head_opt = function x::_ -> Some x | _ -> None
```

Ceci permet de rendre le cas d'erreur plus explicite et *impose* un filtrage pour utiliser la fonction. La version avec exception peut être plus légère d'utilisation car le filtrage d'exception n'est pas nécessaire, mais son absence implique précisément un risque d'erreur à l'exécution (à moins que l'on puisse prouver que l'on n'appelle jamais la fonction avec un argument vide), qui n'est jamais souhaitable.