TD #5 — Un peu d'OCaml (avec solutions)

Exercice 1. *Inférence de types*
Donnez les **types OCaml** de chacune des expressions ci-dessous :

1. `12`

   ```
   int
   ```

2. `false`

   ```
   bool
   ```

3. `()`

   ```
   unit
   ```

4. `let x = 6 in 2 * x`

   ```
   int
   ```

5. `let x = 12 in true`

   ```
   bool
   ```

6. `if false then ()`

   ```
   unit
   ```

7. `fun x -> x`

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

8. `fun x -> 2 * x`

   ```
   int -> int
   ```

9. `fun x y -> x`

   ```
   'a -> 'b -> 'a
   ```

10. `fun x y -> x y`

    ```
    ('a -> 'b) -> 'a -> 'b
    ```

11. `fun x y z -> y (x z)`

    Mieux vaut procéder par étape : `x` est nécessairement une fonction à un paramètre, que l'on peut donc typer comme `(x : 'a -> 'b)`. On en déduit `(z: 'a)` et `(y: 'b -> 'c)`, ce qui donne :
    `('a -> 'b) -> ('b -> 'c) -> 'a -> 'c`

12. `fun x y z -> x z (y z)`

    Mieux vaut procéder par étape : `x` est nécessairement une fonction à deux paramètres, que l'on peut donc typer comme `(x: 'a -> 'b -> 'c)`. On a ensuite nécessairement `(z: 'a)` et `(y z: 'b)`, d'où l'on déduit `(y: 'a -> 'b)`, ce qui donne finalement `('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c`.
    Remarques : Cette fonction est parfois connue sous le nom de *combinateur S* ; par la *correspondance preuve-programme*, elle prouve le théorème « de Frege ».

On pourra (si l'on veut) exploiter le fait que dans les écritures de type fonction la flèche est associative à droite : par exemple, `'a -> 'b -> 'c` est implicitement parenthésé comme `'a -> ('b -> 'c)`.

Exercice 2. *Correction de* `fact`
On considère la fonction :

```
let rec fact x = if x = 0 then 1 else x * (fact (x - 1))
```
Assortie des spécifications suivantes :
Si `x` s'évalue à un entier $x \geq 0$ tel que $x! <$ `max_int`, alors `fact x` s'évalue en un entier de valeur $x!$; sinon le comportement est non défini.

Version alternative (plus facile) des spécifications : comme ci-dessus mais en considérant que `fact` est définie sur un type « `nat` » permettant de représenter exactement tous les éléments de $\mathbb{N}$.

On note $\mathcal{I}$ l'intervalle contenant les valeurs x telles que le comportement de `fact x` est défini.

0. Montrez que si $x+1 \in \mathcal{I}$, alors soit `x` s'évaluant à x et `fx` s'évaluant à $x!$, le calcul du produit `(x + 1) * fx` ne provoque pas de dépassement de capacité.

Par l'hypothèse $x+1 \in \mathcal{I}$ l'on a $(x+1)! <$ `max_int`. Par les propriétés de la factorielle l'on a également $x! \in \mathcal{I}$ et $x \leq x!$ pour tout $x \in \mathbb{N}$, et il s'ensuit que `fx` et `(x + 1)` sont aussi inférieurs à `max_int`. Enfin, par la définition de la factorielle $(x+1)! = (x+1) \times x!$, et donc `(x + 1) * fx` s'évalue correctement à $(x+1)!$ sans provoquer de dépassement de capacité.

1. Montrez la terminaison de `fact` lorsque son argument s'évalue à $x \in \mathcal{I}$.
Vous pouvez procéder directement (par exemple par récurrence), ou utiliser un *variant récursif* (qui fonctionne exactement comme un variant de boucle, *mutatis mutandis*).

Dans tous les cas on a que $\mathcal{I}$ est de la forme $[\![0, m]\!]$ pour un certain entier naturel m, et il n'y a rien à montrer pour $x \in \mathcal{I}$.

Preuve directe par récurrence
Pour $x \in \mathcal{I}$ on note $\mathcal{P}(x)$: `fact x` termine.
Initialisation ($\mathcal{P}(0)$) : La terminaison de `fact 0` est évidente.
Hérédité : On suppose $\mathcal{P}$ vraie au rang x. Si $x+1 \notin \mathcal{I}$ il n'y a rien à montrer. Sinon, de part le code de `fact`, `fact (x + 1)` termine ssi. `fact x` termine, ce qui est vrai par hypothèse de récurrence. $\mathcal{P}$ reste vraie au rang $x+1$.
Conclusion : Par le principe de récurrence, `fact` termine pour tout $x \in \mathcal{I}$.

Preuve par variant récursif
On va montrer que `x` est un variant récursif de `fact`.
Minoration : Soit $x \in \mathcal{I}$, de par le code de `fact` celle-ci n'émet un appel récursif que si $x > 0$; x est minoré par une constante.
Stricte décroissance : Soit $x \in \mathcal{I}$, si `fact` émet un appel récursif elle le fait avec une valeur $x' = x - 1 < x$.
La présence de ce variant garantit la terminaison de `fact`.

2. Montrez sa correction totale relativement à ses spécifications.
Indications : procédez par récurrence, et n'oubliez pas (dans la version non facile) de traiter le cas des (éventuels) dépassement de capacité du type `int`.

On a déjà montré la terminaison pour les valeurs d'argument de `fact` où elle est bien définie, et il suffit donc de prouver sa correction partielle.
Dans ce qui suit on utilise une `police à taille fixe` pour désigner les valeurs OCaml et une *police en italique* pour désigner les entiers naturels, et l'on s'autorisera à mélanger les deux polices dans une même expression.
On procède par récurrence : pour $x \in \mathcal{I}$ on note $\mathcal{P}(x)$: `fact x` s'évalue en $x!$.
Initialisation ($\mathcal{P}(0)$) : On a que `fact 0` s'évalue en `1` (évident), ce qui est bien égal à $1 = 0!$.
Hérédité : On suppose $\mathcal{P}$ vraie au rang x. Si $x+1 \notin \mathcal{I}$ il n'y a rien à montrer. Sinon l'on sait que l'évaluation de `fact (x + 1)` termine et l'on a :

`fact (x + 1)`	=	`(x + 1) * (fact x)`	définition de `fact`
	=	`(x + 1) * x!`	hypothèse de récurrence
	=	$(x+1) \times x!$	$x+1 \in \mathcal{I}$ et **Q.0**
	=	$(x+1)!$	définition de la factorielle

Conclusion : Par le principe de récurrence `fact x` s'évalue en $x!$ pour tout $x \in \mathcal{I}$. Elle est donc (totalement) correcte pour ses spécifications.

Exercice 3. *Calcul du PGCD*

On rappelle que le PGCD (noté $a \wedge b$) de deux entiers naturels a et b satisfait les propriétés suivantes :

— $a \wedge b = b \wedge a$
— $a \wedge 0 = a$
— $a \wedge b = b \wedge r$, où $a \geq b > 0$ et r dénote le reste de la division euclidienne de a par b.

1. Écrivez une fonction `gcd` de type `int -> int -> int` qui pour deux arguments positifs s'évalue en leur PGCD.

En utilisant la construction & déconstruction de types produits (pas encore vue en cours), on propose :

```
let rec gcd a b =
    let a, b = if a > b then a, b else b, a in
    if b = 0 then a else gcd b (a mod b)
```

On admet l'existence en OCaml d'un type `int * int * int` des *triplets* d'`int`, manipulable notamment *via* les syntaxes suivantes :

— `x, y, z` est une expression de type `int * int * int` ssi. `x`, `y` et `z` sont des expressions de type `int`
— soit `t` de type `int * int * int`, `let x, y, z = t` (éventuellement `in`) permet de lier les identifiants `x`, `y` et `z` aux trois éléments de `t`.

Ainsi l'on a par exemple :

```
utop # let t = 1, 2, 3;;
val t : int * int * int = (1, 2, 3)
utop # let x, y, z = t;;
val x : int = 1
val y : int = 2
val z : int = 3
```

On souhaite utiliser (entre autres) ce type pour calculer les coefficients « de Bézout » de deux entiers naturels en plus de leur PGCD, c'est à dire deux entiers (pas nécessairement naturels) u et v tels que $au + bv = a \wedge b$. Pour cela, on utilisera le fait que si $a = qb + r$ (pour $q, r \in \mathbb{N}$), alors :

$$ub + vr = d \Rightarrow (u - vq)b + v(qb + r) = d$$

On admettra le fait que pour un algorithme bien implémenté (tel que celui attendu), les coefficients calculés sont tels que $|u| \leq b/d$ et $|v| \leq a/d$, et donc qu'aucun problème de dépassement de capacité ne se pose.

3. Écrivez une fonction `xgcd` de type `int -> int -> int * int * int` telle que pour $a, b \geq 0$ l'on a que `xgcd a b` s'évalue en le triplet `d, u, v` avec `d` le PGCD de a et b et `u` et `v` les coefficients « de Bézout » associés (dans le « bon ordre »).

Il suffit d'appliquer récursivement la relation donnée dans l'énoncé, en prenant garde à bien retenir s'il y a eu un échange entre les arguments afin de ne pas mal faire correspondre les coefficients calculés.
On propose :

```
let rec xgcd a b =
    let a, b, swpd = if a > b then a, b, false else b, a, true in
    if b = 0
        then (if swpd then a, 0, 1 else a, 1, 0)
        else
            let q, r = (a / b), (a mod b) in
            let d, u', v = xgcd b r in
            if swpd then d, (u' - v * q), v else d, v, (u' - v * q)
```

4. Écrivez une fonction de test `xgcd_test` de type `int -> int -> bool` qui vérifie que `xgcd` calcule correctement le PGCD et les coefficients « de Bézout » de ses arguments.

Il suffit de vérifier que le PGCD divise les arguments et que l'on a bien la relation attendue. On propose :

```
let xgcd_test a b =
    let d, u, v = xgcd a b in
    a mod d = 0 && b mod d = 0 && a*u + b*v = d
```