TD #5 — Un peu d'OCaml

Exercice 1. *Inférence de types*

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

1. `12`
2. `false`
3. `()`
4. `let x = 6 in 2 * x`
5. `let x = 12 in true`
6. `if false then ()`
7. `fun x -> x`
8. `fun x -> 2 * x`
9. `fun x y -> x`
10. `fun x y -> x y`
11. `fun x y z -> y (x z)`
12. `fun x y z -> x z (y z)`

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.
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*).
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`.

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.

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 »).

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.