TP #10 — Premier contact avec OCaml

Exercice 1. Habiter *des types 1*

Écrivez des expressions (les plus simples possibles et qui ne trichent pas (en cas de doute sur ce qui triche : demandez moi)) ayant pour type :

1. `int`
2. `bool`
3. `unit`
4. `int -> int`
5. `int -> bool`
6. `int -> unit`
7. `unit -> int`
8. `int -> int -> int`
9. `'a -> int`
10. `'a -> 'a`
11. `'a -> 'b -> 'a`
12. `'a -> 'b -> 'b`
13. `('a -> 'a) -> 'a -> 'a`
14. `'a -> ('a -> 'b) -> 'b`
15. `('a -> 'b) -> ('b -> 'c) -> 'a -> 'c`
16. `('a -> 'a -> 'b) -> 'a -> 'b` (un peu plus dur)

Afin de facilement vérifier vos propositions (et ne pas les perdre dans les abîmes d'un interpréteur interactif), on conseille de procéder ainsi :

— répondre rapidement aux trois premières questions directement dans un interpréteur interactif ;
— pour les questions suivantes, télécharger puis modifier le fichier mp2i_tp10_ex1_strip.ml, en remplaçant les définitions des identifiants correspondant aux noms des questions par vos réponses (c'est à dire, remplacer à chaque fois `fun _ -> failwith "OHAI"` par votre réponse). On ne demande pas pour l'instant de chercher à comprendre ce que fait cette expression, mais :

17. (Facultatif.) Essayez de comprendre ce que peut faire l'expression :

    ```
    fun _ -> failwith "OHAI"
    ```

    et en quoi elle est utile dans le fichier fourni (et constitue une triche dans le cadre de l'exercice)
    Indice : essayez d'abord d'en inférer le type (vous pouvez ensuite vérifier celui-ci *via* par exemple un interpréteur interactif).

18. Écrivez une expression dont le type est distinct de tous ceux obtenus aux questions précédentes.

Exercice 2. *Fonctions récursives 1*

On rappelle que la syntaxe pour définir (globalement) une fonction récursive en OCaml est :

```
let rec ma_fonction <paramètres> =
    <expression définissant la fonction>
```

1. Donnez le type de la fonction `fact` ci-dessous, et expliquez ce qu'elle calcule :

```
let rec fact x = if x = 0 then 1 else x * (fact (x - 1))
```

2. Que se passe-t'il si `fact` est appelée avec un argument strictement négatif ? Proposez une solution à ce problème (qui peut prendre la forme que vous souhaitez : spécifications précisées, modification du code…)

Une fonctionnalité pratique de `utop` permet de *tracer* les appels de fonction (pas nécessairement récursifs) ; pour cela, il suffit (pour l'exemple de la fonction `fact`) d'entrer la commande :

```
utop # #trace fact
```

On peut également cesser de trace une fonction avec :

```
utop # #untrace fact
```

3. Tracez `fact` puis appelez-la avec de petits arguments (pour lesquels elle se comporte bien), et observez ce qui se passe.

La suite « de Fibonacci » (répertoriée comme A000045 dans l'OEIS) calcule l'effectif d'une population de lapins au cours du temps, dans un modèle idéalisé. Elle est définie récursivement comme :

— $F_0 = 0$
— $F_1 = 1$
— Pour tout $n > 1$, $F_n = F_{n-1} + F_{n-2}$

4. Écrivez une fonction `fib : int -> int` (la plus naïve possible) telle que pour $n \geq 0$, `fib n` s'évalue à F_n (modulo les réductions (« *wraparounds* ») causées par d'éventuels dépassements de capacité du type `int`).
5. Testez.
6. Tracez les appels à `fib` dans `utop`, et observez ce qu'il se passe pour une application à de *petits* arguments. Pensez-vous que votre fonction soit très efficace ?

N.B. Nous verrons prochainement comment analyser le coût d'une fonction récursive comme `fib`, et aussi comment calculer la même chose plus efficacement (jusqu'à doublement exponentiellement mieux, en trichant un peu !)

Exercice 3. *Fonctions récursives 2 : exponentiation entière*

1. Écrivez une fonction `sloexp : int -> int -> int` telle que pour $a, n \in [\![0, 2^{62}-1]\!]$ et $a^n < 2^{62}$, `sloexp a n` s'évalue en a^n. On cherchera à écrire une fonction la plus simple possible.
2. Testez, et constatez notamment l'effet d'un dépassement de capacité.
3. De même pour une fonction `sloexpmod : int -> int -> int -> int` telle que pour $a, n, m \in [\![0, 2^{31}-1]\!]$, `sloexp a n m` s'évalue en $r \in [\![0, m-1]\!]$ tel que $a^n \equiv r \mod m$.
4. Testez.

On souhaite écrire des versions plus efficaces des deux fonctions ci-dessus, en utilisant le fait que pour $a, n > 0$ l'on a $a^{2n} = (a^n)^2$ et $a^{2n+1} = a \times (a^n)^2$.

5. Écrivez une fonction `fastexp : int -> int -> int` de mêmes spécifications que `sloexp`. Cette fonction **ne devra effectuer qu'un seul appel récursif** dans chaque branche de son `if` (pensez à utiliser un `let in` !).
   N'hésitez pas à me solliciter si cette consigne n'est pas claire.
6. Testez, tracez (il est (presque) suffisant de se concentrer sur la dernière partie de la trace).
7. De même pour une fonction `fastexpmod : int -> int -> int -> int` de mêmes spécifications que `sloexpmod`.
8. Testez, tracez.
9. Écrivez une variante `slofastexpmod` de votre fonction `fastexpmod` qui effectue deux appels récursifs dans chaque branche de son `if`.
10. Testez, tracez, et comparez (notamment) les performances avec celles de vos fonctions `sloexpmod` et `fastexpmod`.

Exercice 4. *Fonctions récursives 3 : récursion mutuelle*

(Au moins) deux fonctions f_1 et f_2 sont dites *mutuellement récursives* si f_1 peut récursivement appeler f_2 (et f_1) et f_2 peut récursivement appeler f_1 (et f_2).
OCaml permet de définir de telles fonctions avec la syntaxe :

```
let rec f1 <paramètres> =
    <expression définissant la fonction f1>
and f2 <paramètres> =
    <expression définissant la fonction f2>
```

1. Écrivez deux fonctions :
   - `even : int -> int`
   - `odd ; int -> int`

   mutuellement récursives qui décident si leur argument strictement positif est respectivement pair ou impair.
   Ces fonctions *doivent* être extrêmement naïves et peu efficaces (sinon ce n'est pas très rigolo/intéressant).
2. Testez, tracez.