TP #12 — Troisième contact avec OCaml

Exercice 1. *Entiers naturels* à la Peano

On définit le type OCaml :

```
type nat = O | S of nat
```

Celui-ci permet de représenter (sous forme de valeurs de type `nat`) les entiers naturels (les éléments de $\mathbb{N}$) d'une façon similaire à celle utilisée dans l'axiomatisation de l'arithmétique « de Peano » : l'entier naturel $0 \in \mathbb{N}$ est construit comme `(O : nat)`, et pour tout $x \in \mathbb{N}$, soit `(x : nat)` une représentation de x (que l'on suppose unique), l'unique représentation de $x + 1 \in \mathbb{N}$ est construite comme `(S x : nat)`.

1. Définissez (par une liaison globale) une valeur `zero : nat` qui s'évalue à `O`.
2. Écrivez une fonction OCaml `succ : nat -> nat` qui s'évalue en la représentation du successeur de l'entier représenté par son argument.
3. Écrivez une fonction OCaml `int2nat : int -> nat option` qui s'évalue à `None` si son argument `x` est strictement négatif, et sinon en `Some` de son unique représentation de type `nat` (en assimilant les entiers `int` à des éléments de $\mathbb{Z}$).
4. Écrivez une fonction OCaml `equal : nat -> nat -> bool` telle que `equal x y` s'évalue à `true` si `x` et `y` représentent le même entier, et à `false` sinon.
5. Testez votre fonction `equal` *via* une fonction de test dédiée.
6. Écrivez une fonction OCaml `add : nat -> nat -> nat` telle que `add x y` s'évalue en la représentation de $x + y$, où x et y désignent respectivement les représentations de `x` et `y`.
7. Testez votre fonction `add` *via* une fonction de test dédiée.
8. Écrivez une fonction OCaml `sub : nat -> nat -> nat option` telle que `sub x y` s'évalue en `Some r` avec `r` la représentation de $x - y$ (où x et y désignent toujours respectivement les représentations de `x` et `y`) quand $x - y \geq 0$, et `None` sinon.
9. Testez votre fonction `sub` *via* une fonction de test dédiée.
10. Écrivez une fonction OCaml `mul : nat -> nat -> nat` telle que `mul x y` s'évalue en la représentation de $x \times y$, où x et y désignent respectivement les représentations de `x` et `y`.
    Il peut être utile de d'abord écrire une fonction auxiliaire.
11. Testez votre fonction `mul` *via* une fonction de test dédiée.
12. Que pensez-vous de l'efficacité des calculs effectués dans cette représentation ?
13. Quel lien peut-on faire entre le type `nat` défini ici et le type `unit list` ?

Exercice 2. *~~All the~~ iterators 2 :* `fold` ⋆

Les fonctions *fold* jouent un rôle important en programmation fonctionnelle dans la manipulation des types récursifs (comme les `list`). Le principe d'une telle fonction est de se substituer aux constructeurs de type dans une exploration récursive des données, afin de calculer une valeur à partir des éléments du type et d'une ou plusieurs éventuelles valeurs initiales (se substituant dans ce cas aux constructeurs de base).

Dans le cas des listes, ce principe donne deux fonctions *fold* : l'une « à gauche » et l'une « à droite » (le « vrai » *fold*, dans un sens que l'on ne détaillera pas ici), illustrées à la Fig. 1.

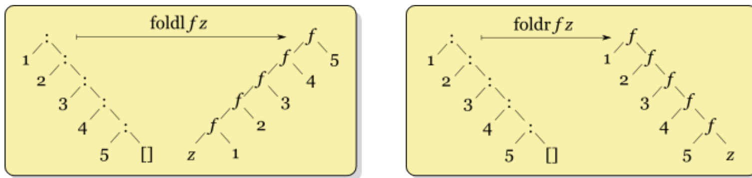

FIGURE 1 – *fold* gauche & droit ; schémas par Cale Gibbard

Plus concrètement, on définit deux fonctions *fold* sur les listes :

— `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
— `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

telles que `fold_left f init [a1; ...; an]` s'évalue en `f (... (f (f init a1) a2) ...) an`, `fold_left f init []` s'évalue en `init` ; `fold_right f [a1; ...; an] init` s'évalue en `f a1 (f a2 (... (f an init) ...))`, `fold_right f [] init` s'évalue en `init`.

1. Écrivez une fonction `fold_left` satisfaisant les spécifications ci-dessus.
2. Testez avec précaution.
3. Écrivez une fonction `fold_right` satisfaisant les spécifications ci-dessus.
4. Testez avec précaution.
5. Utilisez l'une de vos fonctions `fold` pour calculer le produit des éléments d'une liste d'entiers. Le type de fonction (gauche ou droite) a-t'il une importance ici ?
6. Utilisez un `fold` pour écrire une fonction `length : 'a list -> int` telle que `length x` s'évalue en la longueur de `x` (son nombre d'éléments).
7. Utilisez un `fold` pour écrire une fonction `rev` de mêmes spécifications qu'au TP précédent. Celle-ci devra avoir un coût linéaire en la longueur de son argument.
8. Utilisez un `fold` pour écrire une fonction `map'` de même spécifications qu'au TP précédent.

9. Utilisez un `fold` pour écrire une fonction `filter'` de même spécifications qu'au TP précédent.
10. Utilisez un `fold` pour écrire une fonction :

    `flatten : 'a list list -> 'a list`

    telle que pour `x` une liste de listes `[x1 ; ... ; xn]`, `flatten x` s'évalue en la liste obtenue en concaténant `x1...xn` dans cet ordre (c'est à dire la liste `x1 @ ... @ xn`).

Le tri par insertion est un algorithme de tri (pas particulièrement intéressant) dont le principe peut se résumer ainsi :

- une liste vide est triée
- soit une liste triée `x` et un élément `e`, on peut aisément calculer une liste triée `x'` contenant les même éléments que `x` auxquels on a ajouté `e`

11. Écrivez une fonction `insert : 'a -> 'a list -> 'a list` telle que `insert e x` s'évalue en une liste `x'` satisfaisant les spécifications ci-dessus. Bien qu'il soit possible de le faire, *vous n'êtes pas obligés d'utiliser un* `fold` *pour cette fonction*.
12. Utilisez un `fold` pour écrire une fonction `sort : 'a list -> 'a list` qui utilise un tri par insertion pour s'évaluer en une liste triée contenant les même éléments que son argument (avec multiplicité).

Exercice 3. *Quelques fonctions sur des* `Options` ⋆

Le but de cet exercice est de réimplémenter (puis utiliser) quelques unes des fonctions du module `Option`, qui opèrent sur des valeurs de type `'a option`. **Aucune de ces fonctions n'est au programme, et vous n'êtes pas autorisés à les utiliser en général.** Il est également peu probable qu'elles vous soient utiles dans le contexte scolaire des deux prochaines années.

1. Définissez (par une liaison globale) une valeur `none : 'a option` qui s'évalue à `None`.
2. Écrivez une fonction `some : 'a -> 'a option` telle que `some v` s'évalue à `Some v`.
3. Écrivez une fonction `value : 'a option -> 'a -> 'a` telle que `value o d` s'évalue à `v` si `o` s'évalue à `Some v`, et à `d` sinon.
4. Écrivez une fonction `join : 'a option option -> 'a option` telle que `join oo` s'évalue à `v` si `oo` s'évalue à `Some (Some v)`, et à `None` sinon.
5. Écrivez une fonction :

    `map : ('a -> 'b) -> 'a option -> 'b option`

    telle que `map f o` s'évalue à `Some (f v)` si `o` s'évalue à `Some v`, et à `None` sinon (c'est à dire s'il s'évalue à `None`).
6. Écrivez une fonction :

    `bind : 'a option -> ('a -> 'b option) -> 'b option`

telle que `bind o f` s'évalue à `f v` si `o` s'évalue à `Some v`, et à `None` sinon (c'est à dire s'il s'évalue à `None`).

L'utilité de ces fonctions n'est pas forcément immédiatement apparente ; pour tenter de l'illustrer, on considère le cas d'une fonction de division sur les `int` pour laquelle on identifie (ou définit) deux cas d'échecs possibles :

— si le diviseur est nul le calcul de la division provoque une erreur à l'exécution (*viz.* : `Exception: Division_by_zero`) ;

— si le dividende est égal à `min_int` et le diviseur à `-1`, le résultat du calcul est bien défini du point de vue du langage OCaml mais n'est pas égal au résultat de la division sur $\mathbb{Z}$ (pour l'injection naturelle des `int` dans $\mathbb{Z}$) ; autrement dit, on souhaite (*ici*) traiter un dépassement de capacité comme une erreur.

On veut alors définir une fonction `ckd_div` qui s'évalue à `None` si l'un des deux cas d'erreur ci-dessus est rencontré lors du calcul, et `Some v` dans le cas contraire (avec `v` le résultat de la division). On souhaite aussi que cette fonction puisse-t'être utilisée « au milieu » d'une expression plus complexe dont d'autre sous-expressions peuvent échouer (par exemple, on pourrait vouloir enchaîner plusieurs divisions, ou composer différentes opérations arithmétiques pour lesquelles un dépassement de capacité est traité comme une erreur, etc.). Une façon de procéder est d'utiliser des paramètres de type `int option` pour `ckd_div`, et de la faire s'évaluer à `None` quand c'est le cas d'au moins l'un de ses deux arguments. Pour résumer, on souhaite écrire une fonction :

`ckd_div : int option -> int option -> int option`

telle que `ckd_div x y` s'évalue à `None` si `x` ou `y` s'évalue à `None` ou si l'on se trouve dans l'un des deux cas d'erreur ci-dessus, et à `Some v` sinon, pour `v` le résultat (bien défini) de la division `x / y`

7. Écrivez une telle fonction `ckd_div` en utilisant des constructions `match with` et sans utiliser aucune des fonctions implémentées dans les questions précédentes.
8. Écrivez maintenant une seconde version `ckd_div2` de cette fonction qui utilise votre fonction `bind` ci-dessus, mais aucun `match with` explicite (ni aucune de ses variantes syntaxiques). Cette version vous semble-t'elle plus lisible ?

   En pratique, on pourrait encore améliorer (?) la lisibilité en définissant de nouveaux opérateurs, mais c'est une autre histoire...

N.B. Lorsqu'il est doté des fonctions ci-dessus (notamment `bind`), le type `'a option` est un exemple de *monade*. De façon générale, les monades peuvent être utilisées pour propager un état opaque lors d'un calcul purement fonctionnel, ce qui peut notamment servir à gérer des erreurs à l'exécution (comme dans le cas de `ckd_div`).

Exercice 4. Habiter *des types 3* ⋆

Soit le type :

```
type ('a, 'b) either = L of 'a | R of 'b
```

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. `'a -> ('a, 'b) either`
2. `'b -> ('a, 'b) either`
3. `'a * 'b -> ('a, 'b) either`
4. `('a -> 'b) -> ('a, 'b) either -> 'b`
5. `('a -> 'b) -> ('a, 'c) either -> ('b, 'c) either`
6. `('a -> 'b) -> ('c -> 'b) -> ('a, 'c) either -> 'b`
7. `('a, 'b * 'c) either -> ('a, 'b) either * ('a, 'c) either`
8. `('a, 'b) either * ('a, 'c) either -> ('a, 'b * 'c) either`
9. `(('a, 'b) either -> 'c) -> 'a -> 'c`
10. `(('a, 'b) either -> 'c) -> 'b -> 'c`
11. `(('a, 'b) either -> 'c) -> ('a -> 'c) * ('b -> 'c)`
12. `(('a, 'a -> 'b) either -> 'b) -> 'b`
    Pas si simple..? (vous pouvez éventuellement essayer d'utiliser la fonction de la question précédente, mais ce n'est pas nécessaire). En termes de logique, une fonction d'un tel type correspond à une preuve du « tiers-exclu doublement nié ».

Ou un type plus **général**. (En cas de toute sur cette consigne, appelez moi.)

Afin de facilement vérifier vos propositions (et ne pas les perdre dans les abîmes d'un interpréteur interactif), on conseille de télécharger puis modifier le fichier mp2i_tp12_ex4_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).