TP #18 — Arbres cachés & multiplication de polynômes

Exercice 1. *Entiers naturels pairs & impairs*

On définit les types *mutuellement récursifs* suivants :

```
type evenNat = Zero
             | SumEE of evenNat * evenNat
             | SumOO of oddNat * oddNat
             | ProdEE of evenNat * evenNat
             | ProdOE of oddNat * evenNat
             | ProdEO of evenNat * oddNat
and oddNat = One
           | SumOE of oddNat * evenNat
           | SumEO of evenNat * oddNat
           | ProdOO of oddNat * oddNat
```

Ainsi que :

```
type nat = Even of evenNat | Odd of oddNat
```

L'objectif est d'utiliser ces types pour *syntaxiquement* (c'est à dire, par le typage) distinguer un entier naturel pair d'un entier naturel impair :

```
let iseven = function Even _ -> true | _ -> false
let isodd = function Odd _ -> true | _ -> false
```

Pour que ceci ait un intérêt (??) il faut pouvoir doter les expressions de type `nat` d'une *sémantique* cohérente avec celle des entiers naturels usuels, que l'on va pour simplifier assimiler aux valeurs de type `int`). (Aussi, on ignorera dans tout ce qui suit tout problème lié aux éventuels dépassements de capacité sur ce type.)

1. Écrivez deux fonctions OCaml mutuellement récursives :
   ```
   enat2int : evenNat -> int
   onat2int : oddNat -> int
   ```
   qui associent à leur argument l'expression de type `int` leur correspondant « naturellement ». Par exemple, `enat2int (SumOO (One, One))` doit s'évaluer en `2`. *N'hésitez pas à me solliciter si cette consigne n'est pas claire.*
2. Construisez à la main quelques expressions de type `evenNat` et `oddNat` s'évaluant à de petits `int`s (par exemple de `0` à `10`) pour vos fonctions de la question précédente.
3. Représentez ces expressions sous forme d'arbre.
4. Testez.
5. Déduisez de `enat2int` et `onat2int` une fonction OCaml :
   ```
   nat2int : nat -> int
   ```

6. Testez.
7. Écrivez deux fonctions OCaml :

   ```
   natadd : nat -> nat -> nat
   natmul : nat -> nat -> nat
   ```

   qui implémentent l'addition et la multiplication entre deux expressions de type `nat`. Ces fonctions doivent être correctes pour les sémantiques de ces opérations sur les entiers naturels, relativement à l'interprétation implémentée par *nat2int*. Autrement dit, pour tout n_1, n_2 de type `nat`, soit `nu1` et `nu2` de type `int` les résultats de `nat2int n1` et `nat2int n2` respectivement, on doit avoir `nat2int (natadd n1 n2)` (resp. `nat2int (natmul n1 n2)`) égal à `nu1 + nu2` (resp. `nu1 * nu2`).
8. Testez.
9. Écrivez une fonction OCaml `int2nat : int -> nat` qui lève une exception adaptée si son argument est négatif, et sinon construit une expression de type `nat` telle que `nat2int (int2nat n)` s'évalue en n. **Cette expression devra avoir une taille linéaire en celle de n.**
10. Testez.
11. Déduisez-en une fonction OCaml `compress : nat -> nat` telle que `compress n` s'évalue en un `(n' : nat)` s'interprétant identiquement à `n` (par `nat2int`) et de taille de représentation minimale (à un facteur constant près).
12. Quel est le coût de `compress` ? Peut-on faire mieux ?

Exercice 2. *Multiplication rapide de polynômes*

Cet exercice n'est pas extrêmement guidé ; n'hésitez pas à introduire toutes les fonctions intermédiaires que vous estimerez être nécessaires. On conseille aussi de commencer « simplement », avant d'éventuellement réfléchir à comment limiter le nombre de calculs et d'allocations mémoires nécessaires.

Le but de cet exercice est d'implémenter l'algorithme « de Karatsuba » pour multiplier deux polynômes de $\mathbb{A} := \mathbb{Z}/2^{32}\mathbb{Z}[X]$ **de même degré**, dont les coefficients sont stockés sur 32 bits. On rappelle que cet algorithme utilise l'approche *diviser pour régner* suivante : soit P_0, P_1, Q_0, Q_1 quatre polynômes de degré $n-1$ tels que l' on souhaite calculer le produit $R := (P_1X^n + P_0) \times (Q_1X^n + Q_0)$, on pose :

— $A := P_0 \times Q_0$;
— $B := (P_0 + P_1) \times (Q_0 + Q_1)$;
— $C := P_1 \times Q_1$;

ce qui permet de calculer $R = CX^{2n} + (B - (A + C))X^n + A$

1. Définissez un type C `struct poly_u` permettant de représenter un polynôme *via* son degré et ses coefficients stockés sur des entiers de type `uint32_t` non signés de 32 bits.
2. Écrivez des fonctions C d'allocation et de libération de signatures :

— `struct poly_u *alloc_poly_u(size_t deg)`
— `void free_poly_u(struct poly_u *p)`

qui permettent respectivement de créer et détruire un objet de type `struct poly_u`.

3. Testez.
4. Écrivez une fonction permettant d'afficher joliment un polynôme.
5. Testez.
6. Écrivez une fonction C `mulpu` (de signature à déterminer) qui calcule naïvement le produit de deux polynômes de $\mathbb{A}$.
7. Testez.
8. Testez les performances et évaluez expérimentalement le comportement asymptotique de `mulpu` sur des polynômes de degrés variables bien choisis.
9. Écrivez une fonction C `mulpuk1` (de signature à déterminer) qui calcule le produit de deux polynômes de même degré en appliquant *une fois* la décomposition ci-dessus (c'est à dire en utilisant `mulpu` pour calculer les termes A, B et C).
10. Testez (notamment en comparant les résultats avec `mulpu`).
11. Testez les performances sur des polynômes de degrés variables bien choisis.
12. Écrivez une fonction C récursive `mulpukr` (de signature à déterminer) qui calcule le produit de deux polynômes de même degré en appliquant la décomposition ci-dessus tant que le degré du résultat est supérieur à un certain seuil `deg_threshold` que vous définirez globalement (par exemple *via* une variable globale en lecture seule).
13. Testez (notamment en comparant les résultats avec `mulpu`).
14. Testez les performances et évaluez expérimentalement le comportement asymptotique de `mulpukr` sur des polynômes de degrés variables bien choisis, ainsi qu'avec des valeurs variables pour `deg_threshold`.
15. Quelle valeur de seuil vous donne les meilleurs résultats ? Celle-ci dépend-elle du degré initial de vos entrés ?

N.B. La dernière fonction implémentée devrait vous permettre de multiplier des polynômes de degré supérieur à 1 000 000 en un temps raisonnable.

Exercice 3. *Maximum & second maximum*

Le but de cet exercice est d'implémenter un algorithme qui calcule les deux plus grands éléments d'une liste en effectuant peu de *comparaisons* (on laisse en suspend la question de l'intérêt d'une telle entreprise). L'idée de cet algorithme part de l'observation suivante : pour trouver le maximum d'une liste, on peut organiser un « tournoi » entre tous les éléments à comparer, où chaque tour de tournoi procède de la façon suivante : on compare chaque éléments de la liste deux à deux et l'on garde uniquement le plus grand de chaque paire (en cas de longueur impaire, on garde également l'élément qui n'est comparé avec personne) ; l'élément maximum dans la liste est alors simplement l'unique élément de la liste

à un seul élément que l'on obtient après un certain nombre tours. On illustre ceci informellement par l'exemple suivant sur des `int list` :

```
(* début *)
[2; 2; 8; 1; 12; 5; 6; 3; 7; 8; 4; 20]
(* après le tour 1 *)
[2; 8; 12; 6; 8; 20]
(* après le tour 2 *)
[8; 12; 20]
(* après le tour 3 *)
[12; 20]
(* après le tour 4 *)
[20]
(* max *)
20
```

1. Montrez que l'algorithme esquissé ci-dessus calcule le maximum d'une liste en temps linéaire en la longueur de son entrée.
2. Combien de comparaisons sont elles nécessaires dans le calcul du plus grand et second plus grand élément d'une liste, quand celui-ci est fait de façon « évidente » ?

On veut maintenant implémenter un algorithme effectuant ce calcul sur une entrée de longueur n en utilisant seulement $(n-1) + k\lceil \log n \rceil$ comparaisons, avec k une petite constante. Pour cela on remarque (informellement) que dans le tournoi de l'algorithme ci-dessus, le second élément le plus grand a nécessairement été comparé avec l'élément maximum (le « vainqueur » du tournoi) : en effet, cet élément « gagne » nécessairement tous ses « duels », sauf avec le maximum, et seule une comparaison avec ce dernier a pu faire qu'il n'a pas « remporté » le tournoi. Il est donc possible de trouver le second élément le plus grand en recherchant le maximum parmi tous les éléments auxquels le vainqueur du tournoi a été comparé, qui sont en nombre égal au nombre de tours, soit $\lceil \log n \rceil$

Conseils d'implémentation. Dans la suite de l'exercice, il peut éventuellement être pratique d'utiliser la fonction de bibliothèque `Option.compare`, ainsi que le renommage au sein d'un filtrage de motif, que l'on illustre sur un exemple :
`| Some v as o -> e`
permet d'utiliser à la fois `v` et `o` comme un identifiant pour `Some v` dans l'expression `e` associée au cas. Ceci évite la lourdeur de l'introduction d'un `let in`, qui reste cependant une alternative valable :
`| Some v -> let o = Some v in e`
Quoique pratique, cette syntaxe n'est cependant pas au programme.

On définit le type suivant pour représenter (le résultat d') un tournoi :

```
type 'a tourney =
    Max of 'a * 'a tourney option * 'a tourney option
```

Avant le début du tournoi, chaque élément `x` est représenté sous la forme `Max (x, None, None)`, ce qui traduit le fait qu'à ce moment on sait uniquement dire que `x` est maximum de lui-même. Ensuite, le résultat d'un duel (non trivial) entre deux éléments `Max (x1, _, _) as k1` et `Max (x2, _, _) as k2` est représenté comme `Max (x1, Some k1, Some k2)` si `x1` est supérieur à `x2`, et l'expression symétrique sinon.

3. Écrivez une fonction OCaml :
   `one_round : 'a tourney list -> 'a tourney list`
   qui calcule un tour du tournoi suivant l'algorithme ci-dessus, en utilisant la représentation ci-dessus.
4. Écrivez une fonction OCaml :
   `run_tourney : 'a list -> 'a tourney list`
   qui calcule le tournoi complet entre les éléments de son argument.
   Fonction éventuellement utile : `List.map`.
5. Écrivez une fonction OCaml :
   `_max : 'a tourney list -> 'a`
   telle que sa composition avec `run_tourney` fournit le maximum des éléments de l'argument de cette dernière.
6. Testez.
7. Écrivez une fonction OCaml :
   `_2max : 'a -> 'a tourney list -> 'a option`
   telle que pour `m1` l'élément maximum de `(x : 'a list)` dont le résultat complet d'un tournoi est représenté par `t` l'on a `_2max m1 t` qui s'évalue en `Some m2` avec `m2` le second élément le plus grand de `x` (avec multiplicité, c'est à dire que si `m1` apparaît (au moins) deux fois dans `x`, `m2` doit lui être égal), si celle-ci possède au moins deux éléments.
   On conseille de passer par une fonction auxiliaire récursive qui maintient (une `option` sur) un candidat pour le second maximum et le fait évoluer au fur et à mesure. Il est probablement facile de s'égarer un peu lors de l'écriture de cette seconde fonction, mais une utilisation avisée de filtrage dans les arguments et de motifs imbriqués permet de l'écrire avec un unique `match` et en au plus une dizaine de ligne (*sans* compromis sur l'indentation).
8. Déduisez de ce qui précède une fonction OCaml :
   `max12 : 'a list -> 'a * 'a`
   qui lève une erreur si son argument ne contient pas au moins deux éléments, et sinon s'évalue en la paire ordonnée de son maximum et second maximum.
9. Testez.
10. Instrumentez votre code pour compter le nombre de comparaisons effectuées par `max12`, constatez et soyez content.

11. Votre implémentation fait-elle encore des comparaisons dont on pourrait se passer ? Si oui (ce qui est probablement le cas si vous avez suivi l'énoncé) essayez de l'améliorer.
Indice 1 : le type `'a tourney` peut devenir une simple `'a list`, et le code obtenu est nettement plus simple que la version originale.
Indice 2 :

12. Instrumentez à nouveau votre code, constatez et soyez content.