La suite des continuations

Ce TP poursuit l'étude des continuations débutée au TP précédent, et propose d'illustrer les possibilités qu'offrent ces dernières en terme de contrôle du flot de contrôle d'un programme. Un premier exercice demande d'implémenter un mécanisme similaire à un `break` ou à une levée d'exception de façon purement fonctionnelle (et sans nécessiter d'accès « interne » à l'environnement d'exécution), et un second poursuit cette idée en demandant d'implémenter un *générateur* de solutions (à la demande) d'un problème de *backtracking*.

Recommandation générale : testez intensivement les fonctions écrites, et vérifiez expérimentalement vos réponses aux questions (en modifiant au besoin vos fonctions afin de mieux caractériser leur comportement).

Exercice 1. *Produit des termes d'une liste avec sortie anticipée*

On donne la fonction `prod` suivante qui s'évalue en le produit des entiers contenus dans son argument. Cette fonction est écrite en « style direct » (c'est à dire sans utiliser de continuation) et n'est pas *récursive terminale* (la valeur de retour d'un appel émettant un appel récursif est une expression non triviale en la valeur de retour de ce dernier).

```
let rec prod (l:int list) : int =
    match l with
    | [] -> 1
    | hd::tl -> if hd = 0 then 0 else hd * (prod tl)
```

1. Déterminez l'enchaînement des appels récursifs lors de l'évaluation de `prod [1;2;3]` et `prod [1;0;3]`.
2. Combien de produits sont calculés lors d'un appel à `prod` sur une liste contenant un (unique) `0` en $i^{\text{ème}}$ position (depuis la tête de liste) ?
3. Pensez-vous qu'un appel à `prod` cause une allocation mémoire ? Si oui, celle-ci a-t'elle lieu sur la pile ou sur le tas (pour un environnement d'exécution habituel) ?
4. Écrivez en « style à passage de continuation »* une fonction `prodK` de même spécifications que `prod`, qui utilise une fonction auxiliaire `_prodK (l:int list) (k:int -> 'a) : 'a`, où `k` est une continuation. Cette fonction doit être obtenue en modifiant `prod` « le moins possible ».
5. Déterminez l'enchaînement des appels récursifs lors de l'évaluation de `prodK [1;2;3]` et `prodK [1;0;3]`. Votre fonction `_prodK` est-elle récursive terminale ?

∗. *Continuation-passing style*, ou *CPS*.

6. Combien de produits sont calculés lors d'un appel à `prodK` sur une liste contenant un (unique) zéro en $i^{\text{ème}}$ position (depuis la tête de liste) ?
7. Pensez-vous qu'un appel à `prodK` cause-t'il une allocation mémoire ? Si oui, celle-ci a-t'elle lieu sur la pile ou sur le tas (ou sur les deux) ?
8. De quelle autre façon pourriez-vous modifier `prod` afin d'obtenir une fonction récursive terminale de même spécifications ? L'approche alors suivie vous semble-t'elle plus ou moins générale que la « transformation CPS » ayant donné `prodK` ?
9. Modifiez votre fonction `prodK` pour qu'elle utilise une nouvelle fonction auxiliaire

   ```
   _prodK (l:int list) (k:int -> 'a) (ka:unit -> 'a): 'a
   ```

   où `ka` est une continuation à appeler pour sortir de la fonction de façon anticipée. Lorsque son argument contient un zéro, cette fonction ne doit calculer *aucun* produit. Vérifiez que c'est bien le cas.
10. Décrivez (sans les implémenter) deux autres algorithmes (n'utilisant pas nécessairement de continuations) pour calculer le produit des éléments d'une liste, qui n'effectuent aucun produit si la liste contient un zéro, quand :

    a. la liste peut être parcourue plusieurs fois ;
    b. la liste ne peut être parcourue qu'une unique fois (ceci revient de façon équivalente à se placer dans un cadre où la fonction prend en entrée un *flux* (ou *stream*) de données).

    Comparez avec la version de `prodK` de la question précédente.

Exercice 2. *Un générateur de solution pour le problème du cavalier*

Le *problème du cavalier* en dimension $m \times n$ consiste à trouver un ordre de parcours des cases d'un échiquier de dimension $m \times n$ tel que :

— toute case est parcourue exactement une fois ;
— la $i+1^{\text{ème}}$ case à être parcourue est accessible depuis la $i^{\text{ème}}$ case en utilisant les règles de déplacement du cavalier (♘). De façon équivalente, le problème consiste à trouver un « chemin hamiltonien » dans le graphe défini par les déplacements possibles du ♘. (Pour plus de contexte et de jolies animations, on pourra consulter la page wikipédia sur le sujet.)

Ce problème se résout facilement par *backtracking*, et comporte un très grand nombre de solutions même pour des dimensions modestes, cf. https://oeis.org/A165134 (mais pas *trop* modestes, cf. https://xkcd.com/839/). L'objectif de cet exercice est d'implémenter un *générateur* de solution qui chaque fois qu'il est appelé s'évalue (si possible) en une nouvelle solution au problème (distincte de toutes celles déjà calculées). Ceci peut se faire en modifiant de façon minimale une fonction de résolution par backtracking s'évaluant en *une* solution.

Représentation des données. Dans tous l'exercice on utilisera un type :

```
type cb = int option array array
```

pour représenter les solutions (éventuellement partielles) : une solution au problème en dimension $m \times n$ sera une valeur `b` de type `cb` telle que `b` a m lignes et n colonnes et exactement une case de valeur `Some t` pour $0 \leq t < mn$ (en particulier, `b` n'a aucune case valant `None`), et où la numérotation respecte les contraintes du problème (c'est à dire qu'une case de valeur `Some t` doit être accessible depuis la case de valeur `Some (t-1)` (si elle est définie) par un déplacement de cavalier (pour l'association naturelle entre les cases de `cb` et celles d'un échiquier)). Une *solution partielle* est définie de façon identique à la différence que certaines cases peuvent être à `None` ; les valeurs prises par t doivent rester consécutives, mais en revanche il n'est *pas* nécessaire qu'une solution partielle puisse-t'être *étendue* en une solution (non partielle) qui lui est identique là où elle est définie[†].
On supposera également définies des constantes globales `dimX` et `dimY` égales à m et n respectivement.
On donne ci-dessous un exemple de solution et de solution partielle (qui ne peut pas être étendue en solution complète) en dimension 5×5 :

```
[|[|Some  0; Some 19; Some  8; Some 13; Some  2|];
  [|Some  9; Some 14; Some  1; Some 18; Some 23|];
  [|Some 20; Some  7; Some 22; Some  3; Some 12|];
  [|Some 15; Some 10; Some  5; Some 24; Some 17|];
  [|Some  6; Some 21; Some 16; Some 11; Some  4|]|]

[|[|Some  0; Some 19; Some  8; Some 13; Some  2|];
  [|Some  9; Some 14; Some  1; Some 18; None  |];
  [|Some 22; Some  7; Some 20; Some  3; Some 12|];
  [|Some 15; Some 10; Some  5; None   ; Some 17|];
  [|Some  6; Some 21; Some 16; Some 11; Some  4|]|]
```

1. On donne la fonction `reachable` ci-dessous, qui pour une position (i, j) s'évalue en la liste des positions de l'échiquier accessibles par un ♘.

```
let reachable (i,j:int * int) : (int * int) list =
    let reach_base = [(i-2,j+1) ; (i-2,j-1) ;
                      (i-1,j+2) ; (i-1,j-2) ;
                      (i+1,j+2) ; (i+1,j-2) ;
                      (i+2,j+1) ; (i+2,j-1)]
    in
    let within_bounds (x,y) =
        (x >= 0) && (x < dimX) &&
```

†. On utilisera par la suite la notion d'*extension* de solution partielle précisément dans ce sens : une solution (partielle ou complète) *b'* *étend* une solution partielle *b* ssi. pour toute case de *b* de valeur `Some t`, la case correspondante de *b'* vaut également `Some t`.

```
        (y >= 0) && (y < dimY) in
    List.filter within_bounds reach_base
```

Modifiez cette fonction en une fonction :

`reachable_clear (b:cb) (i,j:int*int) : (int*int) list`

qui filtre additionnellement le résultat tel que renvoyé par `reachable` pour en supprimer les cases de *b* qui ont déjà été visitées (c'est à dire, qui ne sont pas à `None`).

Une *heuristique* pour la résolution du problème du cavalier consiste, depuis une case *c*, à choisir comme prochaine case à visiter celle parmi les cases accessibles depuis *c* depuis laquelle le moins de cases sont accessibles (et libres).

2. Écrivez une fonction :

   `heuristic_reorder (b:cb) (r:(int*int) list) : (int*int) list`

   qui prend en entrée une liste de cases de *b* et s'évalue en la liste des mêmes cases triées par ordre croissant de cases accessibles (et libres) depuis elles-mêmes, ainsi qu'une fonction `reachable_heuristic` qui la compose avec `reachable_clear`. (Il peut être utile d'utiliser ici les itérateurs `List.map` et `List.sort`.)

On rappelle que les types `array` n'étant pas immuables, il se peut que lors d'un appel (notamment récursif) à une fonction avec un `array` *b* comme argument, la valeur de *b* aura été modifiée au retour de l'appel. Prenez bien cela en compte lors de l'écriture de vos fonctions ci-dessous (une approche possible consiste à uniquement modifier les variables de type `cb` par l'intermédiaire d'une fonction `set` qui effectue d'abord une copie profonde de son argument `cb` et renvoie une modification de cette copie).

3. Écrivez une fonction récursive :

   `_kt (b:cb) (dest:(int*int) list) (t:int) : cb option`

   qui prend en entrée une solution partielle *b* avec *t* cases de valeur autre que `None`, une liste *dest* (possiblement vide) de cases accessibles et libres depuis la dernière case visitée, et qui utilise une approche par backtracking pour :

   — s'évaluer à `None` s'il n'existe aucune solution complète qui étend *b* et pour laquelle la case de valeur `Some t` se trouve dans celles données par *dest* ;
   — s'évalue à `Some b'` avec *b'* une telle solution, s'il en existe une.

4. Écrivez une fonction `kt (si:int) (sj:int)` qui utilise `_kt` pour calculer une solution au problème du cavalier commençant à la case *(si, sj)*, et échoue arbitrairement s'il n'en existe pas.
5. Utilisez votre fonction *kt* pour trouver des solutions en dimension carrée 5, 8, 10, 70… Constatez sur de petits cas l'amélioration de performance apportée par l'utilisation de l'heuristique ci-dessus.

6. Écrivez une nouvelle fonction qui modifie *a minima* votre fonction `_kt` en :
   ```
   _ktl (b:cb) (dest:(int*int) list) (t:int)
   (sols:cb list) (lim:int) : cb list
   ```
   qui prend comme arguments supplémentaires une liste de solutions *sols* et une taille limite *lim* pour cette liste, et qui s'il existe des solutions étendant *b* avec *dest* (dans le même sens que pour `_kt`) s'évalue en une liste complétant *sols* avec ces solutions (dans la limite d'une taille totale *lim*), et sinon en *sols*.
7. Écrivez une fonction `ktl (si:int) (sj:int) (lim:int)` qui s'évalue en une liste d'au plus *lim* solutions au problème du cavalier commençant à la case *(si, sj)*.
8. Vérifiez que votre fonction trouve le bon nombre de solutions pour les 25 cases de départ d'un échiquier carré de dimension 5, comme indiqué dans l'exemple donné sur https://oeis.org/A165134.

La fonction `ktl` précédente permet en principe d'énumérer toutes les solutions à une instance du problème du cavalier. Cependant, elle ne fait pas cela de façon « interactive » : un ou une utilisatrice doit par avance choisir un nombre de solution maximum à calculer, et le temps de calcul et la consommation mémoire de `ktl` seront au moins linéaires en le nombre effectivement calculé : il faudra attendre que toutes les solutions ont été calculées avant même d'obtenir la première. Cela n'est pas une bonne nouvelle pour les impatients, d'autant moins qu'une résolution par backtracking n'offre (*a priori*) pas de garantie sur le temps nécessaire pour trouver chaque solution supplémentaire : il serait donc préférable de rendre chaque solution trouvée disponible dès que possible.
Les questions suivantes ont pour objectif de résoudre tous ces problèmes (et plus !) en implémentant un *générateur* de solutions. Ceci se fera en suivant une approche par continuation qui ne modifie `_ktl` que de façon minime.

On définit le type somme suivant :
```
type 'a gen = Done | More of 'a * (unit -> 'a gen)
```
Celui-ci représente un générateur de valeurs de type `'a` distinguant deux cas : `Done` quand il n'y a plus de valeur à générer ; `More(v, g)` où `v` est une valeur qui a été générée et `g` est une fonction pouvant être appelée pour générer (si possible) une *nouvelle* valeur.

9. Écrivez une nouvelle fonction qui modifie votre fonction `_ktl` *a minima* en :
   ```
   _ktc (b:cb) (dest:(int*int) list) (t:int)
   (k : unit -> 'a gen) : 'a gen
   ```
   qui remplace les arguments *sols* et *lim* par une unique continuation `(k : unit -> 'a gen)`. S'il existe des solutions étendant `b` avec `dest` cette fonction s'évalue en une solution parmi celles-ci et une fonction permettant de générer les éventuelles autres solutions, et dans le cas contraire elle s'évalue en `Done`.

10. Écrivez une fonction `ktc (si:int) (sj:int)` qui utilise `_ktc` et permet de *générer à la demande* toutes les solutions au problème du cavalier commençant à la case *(si, sj)*.

11. Testez cette fonction (de préférence dans un interpréteur interactif).

Quand elle ne s'évalue pas en `Done`, la fonction `_ktc` fournit une solution *complète* au problème du cavalier (ainsi qu'un générateur pour les solutions suivantes). Il est cependant facile de la modifier pour qu'elle fournisse (au fur et à mesure) chaque solution partielle visitée lors de la résolution par backtracking. Ceci peut s'assimiler à une exécution *pas à pas* d'un débuggueur, et peut être utile pour du débugguage[‡] ou tout simplement pour visualiser le processus de résolution par backtracking (ce qui est rigolo en soi).

12. Écrivez une fonction `_ktci` de même signature que `_ktc` et qui modifie celle-ci pour qu'elle s'évalue en chaque solution partielle visitée lors de la résolution.

13. Écrivez une fonction `ktci (si:int) (sj:int)` qui utilise `_ktci` pour générer pas à pas les solutions partielles visitées lors d'une résolution du problème du cavalier depuis la case *(si, sj)*.

14. Testez cette fonction (de préférence dans un interpréteur interactif), par exemple en dimension carrée 5 et pour constater l'effet de l'utilisation ou non de l'heuristique. Vous pouvez (si vous le souhaitez) utiliser pour cela la fonction d'« affichage joli » suivante (qui peut d'ailleurs être améliorée pour être encore plus jolie) :

```
let pp (b:cb) : unit =
    let mv = ref b.(0).(0) in
    let mi, mj = ref 0, ref 0 in
    let pr i j v =
        if (i = !mi) && (j = !mj) then
            print_string " K "
        else
            match v with
            | None -> print_string " . "
            | Some t -> print_string " + "
    in
    begin
        for i = 0 to dimX - 1 do
            for j = 0 to dimY - 1 do
                if (Option.compare compare b.(i).(j) !mv)
                    = 1 then
                    (mv := b.(i).(j) ; mi := i ; mj := j)
```

‡. Histoire vraie : c'est en implémentant et testant cette fonction que je me suis rendu compte qu'il y avait une erreur dans ma gestion des `Array` en fin de résolution (ce qui expliquait pourquoi mon générateur trouvait peu de solutions).

```
            done
        done ;
        for i = 0 to dimX - 1 do
            (Array.iteri (pr i) b.(i) ; print_newline ())
        done
    end
```