TD induction structurelle

A. Listes

Exercice 1 (Concaténation de listes) On considère l'ensemble des listes définies par :
- la liste vide [],
- le constructeur ::, qui crée une nouvelle liste $e :: l$ en ajoutant un élément e en tête d'une liste l.

On s'intéresse à la *concaténation* de deux listes l_1 et l_2, c'est-à-dire à l'opération qui forme une unique liste $\text{concat}(l_1, l_2)$ contenant, dans l'ordre, les éléments de l_1 puis ceux de l_2.

1. Définir une fonction longueur : liste $\to \mathbb{N}$ donnant la longueur d'une liste.
2. Définir une fonction concat : liste $\times$ liste $\to$ liste effectuant une concaténation de listes.
3. Démontrer que la concaténation additionne les longueurs, c'est-à-dire que

$$\forall l_1, l_2 \in \text{liste}, \quad \text{longueur}(\text{concat}(l_1, l_2)) = \text{longueur}(l_1) + \text{longueur}(l_2)$$

4. Démontrer que la concaténation est associative, c'est-à-dire que

$$\forall l_1, l_2, l_3 \in \text{liste}, \quad \text{concat}(l_1, \text{concat}(l_2, l_3)) = \text{concat}(\text{concat}(l_1, l_2), l_3)$$

Correction :

1.

$$\begin{aligned} \textit{longueur}([]) &= 0 \\ \textit{longueur}(x :: l) &= 1 + \textit{longueur}(l) \end{aligned}$$

2.

$$\begin{aligned} \textit{concat}([], l_2) &= l_2 \\ \textit{concat}(x :: l, l_2) &= x :: \textit{concat}(l, l_2) \end{aligned}$$

3. Par récurrence sur l_1.

—

$$\begin{aligned} \textit{longueur}(\textit{concat}([], l_2)) &= \textit{longueur}(l_2) \\ &= 0 + \textit{longueur}(l_2) \\ &= \textit{longueur}([]) + \textit{longueur}(l_2) \end{aligned}$$

—

$$\begin{aligned} \textit{longueur}(\textit{concat}(l :: l_2)) &= \textit{longueur}(x :: \textit{concat}(l, l_2)) \\ &= 1 + \textit{longueur}(\textit{concat}(l, l_2)) \\ &= 1 + (\textit{longueur}(l) + \textit{longueur}(l_2)) \\ &= (1 + \textit{longueur}(l)) + \textit{longueur}(l_2) \\ &= \textit{longueur}(x :: l) + \textit{longueur}(l_2) \end{aligned}$$

4. Par récurrence sur l_1.

—

$$\begin{aligned} \textit{concat}([], \textit{concat}(l_2, l_3)) &= \textit{concat}(l_2, l_3) \\ &= \textit{concat}(\textit{concat}([], l_2), l_3) \end{aligned}$$

—

$$\begin{aligned} \textit{concat}(x :: l, \textit{concat}(l_2, l_3)) &= x :: \textit{concat}(l, \textit{concat}(l_2, l_3)) \\ &= x :: \textit{concat}(\textit{concat}(l, l_2), l_3) \\ &= \textit{concat}(x :: \textit{concat}(l, l_2), l_3) \\ &= \textit{concat}(\textit{concat}(x :: l, l_2), l_3) \end{aligned}$$

□

Exercice 2 (Renversement de liste) Voici une définition pour une fonction rev : liste $\to$ liste de renversement de liste.

$$\begin{aligned} \text{rev}([]) &= [] \\ \text{rev}(x :: l) &= \text{concat}(\text{rev(l)}, x :: []) \end{aligned}$$

1. Démontrer que rev préserve la longueur :

$$\forall l \in \text{liste}, \quad \text{longueur}(\text{rev}(l)) = \text{longueur}(l)$$

2. Démontrer que rev se distribue sur concat de la manière suivante :

$$\forall l_1, l_2 \in \text{liste}, \quad \text{rev}(\text{concat}(l_1, l_2)) = \text{concat}(\text{rev}(l_2), \text{rev}(l_1))$$

3. Démontrer que rev est involutive, c'est-à-dire que

$$\forall l \in \text{liste}, \quad \text{rev}(\text{rev}(l)) = l$$

Correction :

1. *Par récurrence sur l.*
   - *Cas [].*

$$\begin{aligned} longueur(rev([])) &= longueur([]) \end{aligned}$$

   - *Cas $x :: l$.*

$$\begin{aligned} longueur(rev(x :: l)) &= longueur(concat(rev(l), x :: [])) \\ &= longueur(rev(l)) + longueur(x :: []) \\ &= longueur(l) + longueur(x :: []) \\ &= longueur(l) + 1 \\ &= longueur(x :: l) \end{aligned}$$

2. *Par récurrence sur l_1.*
   - *Cas [].*

$$\begin{aligned} rev(concat([], l_2)) &= rev(l_2) \\ &= concat(rev(l_2), []) \\ &= concat(rev(l_2), rev([])) \end{aligned}$$

   - *Cas $x :: l$.*

$$\begin{aligned} rev(concat(x :: l, l_2)) &= rev(x :: concat(l, l_2)) \\ &= concat(rev(concat(l, l_2)), x :: []) \\ &= concat(concat(rev(l_2), rev(l)), x :: []) \\ &= concat(rev(l_2), concat(rev(l), x :: [])) \\ &= concat(rev(l_2), rev(x :: l)) \end{aligned}$$

3. *Par récurrence sur l.*
   - *Cas [].*

$$\begin{aligned} rev(rev([])) &= rev([]) \\ &= [] \end{aligned}$$

   - *Cas $x :: l$.*
     Remarquons d'abord que $rev(x :: []) = concat(rev([]), x :: []) = concat([], x :: []) = x :: []$.

$$\begin{aligned} rev(rev(x :: l)) &= rev(concat(rev(l), x :: [])) \\ &= concat(rev(x :: []), rev(rev(l))) \\ &= concat(x :: [], rev(rev(l))) \\ &= concat(x :: [], l) \\ &= concat([], x :: l) \\ &= x :: l \end{aligned}$$

□

Exercice 3 (Correction d'un renversement récursif terminal) La définition précédente de rev est adaptée pour spécifier le renversement d'une liste et démontrer ses propriétés, mais elle est assez éloignée des réalisations réelles. On va maintenant faire le lien entre les deux. Voici la définition d'une fonction rev_rt : liste → liste correspondant à une réalisation récursive terminale et efficace du renversement de liste.

$$\begin{aligned} \text{rev\_rt}(l) &= \text{aux}(l, []) \\ \text{aux}([], a) &= a \\ \text{aux}(x :: l, a) &= \text{aux}(l, x :: a) \end{aligned}$$

Démontrer que rev_rt est correcte, c'est-à-dire que

$$\forall l \in \text{liste}, \quad \text{rev\_rt}(l) = \text{rev}(l)$$

Indication : il faudra démontrer quelque chose par récurrence, mais quoi exactement ?
Correction : On démontre par récurrence sur l que $\forall a,\ aux(l, a) = concat(rev(l), a)$.

- *Cas [].*

$$\begin{aligned} aux([], a) &= a \\ &= concat([], a) \\ &= concat(rev([]), a) \end{aligned}$$

— *Cas* $x :: l$.

$$\begin{aligned} aux(x :: l, a) &= aux(l, x :: a) \\ &= concat(rev(l), x :: a) \\ &= concat(rev(l), concat(x :: [], a)) \\ &= concat(concat(rev(l), x :: []), a) \\ &= concat(rev(x :: l), a) \end{aligned}$$

□

B. Arbres

Exercice 4 (Arbres binaires de recherche) On définit l'ensemble des arbres binaires dont les nœuds sont étiquetés par des entiers avec la signature suivante :

— l'objet de base E représente un arbre vide,

— le constructeur N s'applique à deux arbres t_1 et t_2 et un entier n pour construire un nouvel arbre $N(t_1, n, t_2)$ dont la racine est étiquetée par n et dont les sous-arbres gauche et droit sont respectivement t_1 et t_2.

Dans un premier temps, on s'intéresse à tous les arbres qui peuvent être construits ainsi.

1. Définir une fonction infixe : arbre → liste qui étant donné un arbre binaire t, renvoie la liste composée des entiers contenus dans l'arbre dans l'ordre donné par un parcours infixe (l'entier étiquetant un nœud est donc pris en compte entre les deux sous-arbres de ce nœud).
2. Définir une fonction appartient : $\mathbb{N} \times$ arbre → $\mathbb{B}$ testant l'appartenance d'un entier à un arbre et renvoyant un booléen.

On se concentre maintenant sur les *arbres binaires de recherche*, qui vérifient que dans chaque sous-arbre de la forme $N(t_1, n, t_2)$, tous les entiers stockés dans le sous-arbre t_1 sont inférieurs ou égaux à n qui est lui-même inférieur ou égal à tous les entiers stockés dans le sous-arbre t_2.

3. Les termes suivants sont-ils des arbres binaires de recherche ?
   (a) N(N(F,1,F),2,N(F,3,F))
   (b) N(F,2,N(N(F,1,F),3,F))
4. Écrire deux termes différents qui correspondent à des arbres binaires de recherche contenant les valeurs 1, 2, 3 et 4. Donner la valeur de la fonction infix pour ces arbres.
5. Démontrer que si l'arbre t est un arbre binaire de recherche, alors le mot infixe(t) est trié.

La structure d'arbre binaire de recherche permet de chercher efficacement un élément.

6. Lorsque l'on recherche un élément n dans un arbre binaire de recherche de la forme $N(t_1, m, t_2)$, est-il utile de fouiller l'ensemble de l'arbre ? dans quel cas faut-il chercher dans t_1 et dans quel cas dans t_2 ?
7. Définir une fonction appartient_abr : $\mathbb{N} \times$ arbre → $\mathbb{B}$ telle que inabr(n, t) teste si l'entier n est présent dans l'arbre t, en supposant que t est un arbre binaire de recherche.
8. Montrer que si appartient_abr(n, t) = vrai alors appartient(n, t) = vrai.
9. Montrer que si appartient(n, t) = vrai et t est un arbre binaire de recherche, alors appartient_abr(n, t) = vrai.

On peut également définir une procédure efficace pour vérifier qu'un arbre binaire est bien un arbre binaire de recherche.

10. Définir une fonction verif : $\mathbb{N} \times \mathbb{N} \times$ arbre → $\mathbb{B}$ telle que verif(min, max, t) renvoie vrai si et seulement si t est un arbre binaire de recherche dont tous les entiers x vérifient $min \leq x \leq max$.
11. Démontrer que la fonction verif est correcte.

Comme nous l'avons déjà dit, les termes et fonctions récursives sur les termes sont facilement représentables en caml.

12. Définir en caml un type des arbres binaires, ainsi que les fonctions infix, appartient, appartient_abr et verifie.

Correction :

1. *Un cas pour chaque forme de terme.*

   $$infix(F) = [] \qquad infix(N(t_1, n, t_2)) = concat(infix(t_1), n :: infix(t_2))$$

2. *Disjonction sur les différents endroits où peut se trouver n.*

   $$appartient(n, F) = false \qquad appartient(n, N(t_1, m, t_2)) = appartient(n, t_1) \vee n = m \vee appartient(n, t_2)$$

3. *Le premier oui, le deuxième non (car* 1 *dans le sous-arbre droit du nœud portant l'étiquette* 2*).*
4. *Deux exemples : N(N(N(F,1,F),2,N(F,3,F)),4,F) et N(N(F,1,F),2,N(N(F,3,F),4,F)).*
5. *Récurrence sur t supposé ABR.*
   — *Cas F. On a infix(F) = Nil, triée.*
   — *Cas* $N(t_1, n, t_2)$*. Cet arbre étant un ABR, ses sous-arbres* t_1 *et* t_2 *sont des ABR également. Par hypothèses de récurrence,* $infix(t_1)$ *et* $infix(t_2)$ *sont donc triées. En outre, tous les éléments de* $infix(t_1)$ *(resp.* $infix(t_2)$*) sont inférieurs ou égaux (resp. supérieurs ou égaux) à n (lemme utilisé implicitement ici : tous les éléments de* $infix(t)$ *sont des éléments de t).*
   Donc $n :: infix(t_2)$ *est triée, et donc* $concat(infix(t_1), n :: infix(t_2)) = infix(N(t_1, n, t_2))$ *est triée également.*
6. *On cherche à gauche (resp. à droite) lorsque* $n < m$ *(resp.* $m < n$*). Pas besoin de chercher à gauche ni à droite si* $n = m$*.*
7. *On sépare le cas N en trois pour tenir compte de scritères de la question précédente.*

$$\begin{aligned} appartient\_abr(n, F) &= false \\ appartient\_abr(n, N(t_1, m, t_2)) &= true && si\ n = m \\ appartient\_abr(n, N(t_1, m, t_2)) &= appartient\_abr(n, t_1) && si\ n < m \\ appartient\_abr(n, N(t_1, m, t_2)) &= appartient\_abr(n, t_2) && si\ m < n \end{aligned}$$

8. *Par récurrence sur t.*
   - *Cas F. $appartient\_abr(n, F) = false$, rien à traiter.*
   - *Cas $N(t_1, m, t_2)$. Par cas sur les conditions pouvant rendre $appartient\_abr(n, N(t_1, m, t_2))$ vraie.*
     - *Cas $n = m$. Alors $appartient(n, N(t_1, m, t_2)) = true$.*
     - *Cas $n < m \wedge appartient\_abr(n, t_1)$. En particulier $appartient\_abr(n, t_1)$, donc par hypothèse de récurrence $appartient(n, t_1) = true$, donc $appartient(n, N(t_1, m, t_2)) = true$.*
     - *Cas $m < n \wedge \ldots$ similaire.*
9. *Par récurrence sur t.*
   - *Cas F. $appartient(n, F) = false$, rien à faire.*
   - *Cas $N(t_1, m, t_2)$. Par cas sur la disjonction définissant $appartient(n, N(t_1, m, t_2))$ (la formule est vraie si au moins l'une des clauses est vraie).*
     - *Cas $n = m$. Alors $appartient\_abr(n, N(t_1, m, t_2)) = true$.*
     - *Cas $appartient(n, t_1) = true$. Par hypothèse de récurrence, $appartient\_abr(n, t_1) = true$. En outre, $N(t_1, m, t_2)$ étant un ABR et n étant présent dans le sous-arbre gauche, on a $n \leq m$ (et le cas $n = m$ étant déjà traité on peut se concentrer ici sur le cas $n < m$). Donc $appartient\_abr(n, N(t_1, m, t_2)) = true$.*
     - *Cas $appartient(n, t_2) = true$ similaire.*
10.

$$\begin{aligned} verif(m, M, F) &= true \\ verif(m, M, N(t_1, n, t_2)) &= m \leq n \leq M \wedge verif(m, n, t_1) \wedge verif(x, M, t_2) \end{aligned}$$

11. *Récurrence sur t.*
    - *Cas F : correcte, car F est bien un ABR.*
    - *Cas $N(t_1, n, t_2)$. Si $verif(m, M, N(t_1, n, t_2)) = true$ alors nécessairement on a $m \leq n \leq M$, $verif(m, n, t_1)$ et $verif(x, M, t_2)$. Par hypothèses de récurrence on a donc t_1 et t_2 ABR, avec tous éléments n_1 de t_1 tels que $m \leq n_1 \leq n$ et tous éléments n_2 de t_2 tels que $n \leq n_2 \leq M$. Donc d'une part tous les éléments de t_1 sont inférieurs ou égaux à n et tous les éléments de t_2 sont supérieurs ou égaux à n : l'arbre est bien un ABR. Et d'autre part n, et tous les éléments de t_1 et t_2 sont bien compris entre m et M.*
12. *...*

□

C. Programmation

Dans toute cette section, on part de l'interprète de IMP donné dans le cours (et dont le code est redonné sur la page du cours).

Exercice 5 (Opérations unaires) Étendre la syntaxe abstraite et l'interprète de IMP pour y inclure deux opérations unaires : l'opposé (`-`), et la négation (`!`). □

Exercice 6 (Des valeurs plus riches) Modifier l'interprète de IMP pour que le type `value` des résultats produits par `eval` distingue les valeurs entières et les valeurs booléennes. On se donne la définition suivante.

```
type value =
  | Int of int
  | Bool of bool
```

Attention, ceci fera apparaître des situations que l'on peut considérer comme étant des erreurs manifestes (par exemple, une addition de booléens). Dans ce cas, déclencher une erreur.
Indication : une manière élégante de traiter ceci consiste à adjoindre deux fonctions `eval_int: expr -> int` *et* `eval_bool: expr -> bool` *à la fonction principale* `eval: expr -> value`. *Chacune des deux fonctions auxiliaires suppose avoir reçu en argument une expression produisant une valeur du bon type et renvoie son contenu, et échoue dans le cas contraire.* □

Exercice 7 (Paires) On veut étendre les expressions du langage IMP avec trois constructions :
- la paire `(e1, e2)` de deux expressions `e1` et `e2`,
- l'opération `fst(e)` qui, en supposant que la valeur de `e` est une paire, renvoie le premier élément de la paire,
- l'opération `snd(e)` qui, en supposant que la valeur de `e` est une paire, renvoie le deuxième élément de la paire.

Étendre la syntaxe abstraite des expressions pour intégrer ces trois éléments. Cet ajout implique également une nouvelle forme de valeur (une paire de deux valeurs), à ajouter à la définition des valeurs. Étendre enfin l'interprète pour traiter ces nouvelles constructions.
Correction :

```
type uop = Opp | Neg | Fst | Snd
type bop = Add | Mul | Lt | And | Pair
type expr =
  | Cst of int
  | Var of string
  | Uop of uop * expr
  | Bop of bop * expr * expr

type instr =
  | Set  of string * expr
```

```
  | If   of expr * seq * seq
  | While of expr * seq
  | Print of expr
and seq = instr list

type value =
  | Int of int
  | Bool of bool
  | Pair of value * value

let exec_prog (p: seq): unit =
  let env = Hashtbl.create 64 in

  let rec eval_int e = match eval e with
    | Int n -> n
    | _ -> failwith "not an int"
  and eval_bool e = match eval e with
    | Bool b -> b
    | _ -> failwith "not a bool"
  and eval_pair e = match eval e with
    | Pair(v1, v2) -> v1, v2
    | _ -> failwith "not a pair"

  and eval: expr -> value = function
    | Cst n -> Int n
    | Var x -> Hashtbl.find env x
    | Bop(Add, e1, e2) -> Int (eval_int e1 + eval_int e2)
    | Bop(Mul, e1, e2) -> Int (eval_int e1 * eval_int e2)
    | Bop(Lt, e1, e2) -> if eval_int e1 < eval_int e2 then Bool true else Bool false
    | Bop(And, e1, e2) -> if eval_bool e1 = false then Bool false else eval e2
    | Uop(Opp, e) -> Int (- eval_int e)
    | Uop(Neg, e) -> Bool (not (eval_bool e))
    | Uop(Fst, e) -> let v1, _ = eval_pair e in v1
    | Uop(Snd, e) -> let _, v2 = eval_pair e in v2
    | Bop(Pair, e1, e2) -> Pair(eval e1, eval e2)

  and exec: instr -> unit = function
    | Set(x, e) -> let v = eval e in Hashtbl.replace env x v
    | If(e, s1, s2) ->
       if eval_bool e then exec_seq s1 else exec_seq s2
    | While(e, s) as i ->
       if eval_bool e then (exec_seq s; exec i)
    | Print(e) -> let v = eval_int e in Printf.printf "%c\n" (Char.chr v)
  and exec_seq (s: seq): unit =
    List.iter exec s
  in
  exec_seq p
```

□

Exercice 8 (Branchement généralisé) Étendre la syntaxe abstraite et l'interprète du langage IMP avec une notion branchement généralisé de type `switch/case`. On veut notamment ajouter une instruction `switch` comportant :

- une expression discriminante `e`,
- une séquence de branches de la forme `n: {s}` où `n` est un entier à comparer à la valeur de `e` et `s` une séquence d'instructions à exécuter le cas échéant,
- une séquence d'instructions par défaut, à exécuter si aucune branche n'a été sélectionnée.

□

Exercice 9 (Sorties de boucle) Étendre la syntaxe abstraite et l'interprète du langage IMP avec deux instructions `break` et `continue`. Rappels :

- `break` met fin à l'exécution d'une boucle,
- `continue` met fin à l'exécution d'un tour de boucle,

et l'un et l'autre s'appliquent à la boucle la plus interne.

Indication. Voici deux pistes pour inclure dans l'interprète la sémantique de ces deux instructions :

- si vous le connaissez, vous pouvez utiliser le mécanisme des exceptions de caml,
- vous pouvez faire en sorte que les fonctions `exec` et `exec_seq` renvoient une valeur indiquant la manière dont l'exécution s'est terminée, à choisir par exemple entre « normal », « `break` » et « `continue` ».

Il est également intéressant de comparer les deux versions. □