Devoir surveillé 1 c

C.C.P. 2009

Option informatique MP1, MP2 & MP3

Gestion de blocs mémoire

I Représentation des blocs

I.1 Définitions

I.2 Représentation en OCaml

I.3 Opérations sur la structure de bloc

Question 1

On applique la définition.

```
let adjacent b1 b2 =
   let (d1, t1) = b1 in
   let (d2, t2) = b2 in
   d2 = d1 + t1 || d1 = d2 + t2;;
```

On peut décomposer les variables dans les paramètres

```
let adjacent (d1, t1) (d2, t2) =
   d2 = d1 + t1 || d1 = d2 + t2;;
```

Question 2

```
let fusion (d1, t1) (d2, t2) =
   if d2 = d1 + t1
   then (d1, t1 + t2)
   else begin
        if d1 = d2 + t2
        then (d2, t1 + t2)
        else (-1, -1) end;;
```

On peut utiliser ce qui a déjà été écrit

```
let fusion (d1, t1) (d2, t2) =
   if adjacent (d1, t1) (d2, t2)
   then (min d1 d2, t1 + t2)
   else (-1, -1);;
```

II Codage binaire des entiers positifs

II.1 Définitions

Question 3

La question est ambiguë : veut-on une formule mathématique ou un algorithme ? La réponse algorithmique est demandée à la question 5 donc on donne une réponse mathématique.

On suppose $n \geq 1$. Si k est l'indice cherché, on a $n = 2^{k-1} + \sum_{i=1}^{k-2} b_i 2^{i-1}$.

Or $0 \leq \sum_{i=1}^{k-2} b_i 2^{i-1} \leq \sum_{i=1}^{k-2} 2^{i-1} = 2^{k-1} - 1$ donc $.2^{k-1} \leq n \leq 2^{k-1} + 2^{k-1} - 1 = 2^k - 1$ d'où

$$k = \lfloor \log_2(n) \rfloor + 1 = \lfloor \log_2(2n) \rfloor$$

Question 4

C'est la décomposition unique en produit de facteurs premiers.
Si on veut démontrer ce cas particulier, on peut démontrer l'existence par récurrence forte :
$\mathcal{P}(n)$: tout nombre entier compris entre 1 et n s'écrit sous la forme $p.2^k$ avec p impair.
$\mathcal{P}(1)$ est vrai car $1 = 1.2^0$
Si $\mathcal{P}(n)$ est vraie, on a deux possibilité pour $n+1$

- si $n+1$ est impair, alors $n+1 = (n+1).2^0$ convient,
- si $n+1$ est pair, alors $n+1 = 2m$ avec $m < n+1$ donc $m \leq n$ et, d'après l'hypothèse de récurrence, $m = p.2^k$ donc $n+1 = p.2^{k+1}$ avec p impair.

On a ainsi prouvé l'existence.
Si on a $n = p.2^k = q.2^r$, avec q et q' impairs, on peut supposer $k \leq r$.
On a alors $p = q.2^{r-k}$ avec p impair donc $r - k = 0$ puis $p = q$ d'où l'unicité.

II.2 Représentation en OCaml

II.3 Nombre de bits nécessaire

Question 5

n demande un bit de plus que `n/2`, même pour $n = 1$.

```
let rec nombre n =
   match n with
   |0 -> 0
   |n ->   1 + nombre (n/2);;
```

II.4 Codage

Question 6

Les divisions successives par 2 permettent de définir les chiffres binaires avec le bit de parité. Dans une récursivité classique le bit de parité, c'est-à-dire le bit de poids faible sera calculé en dernier et ajouté en tête. On doit retourner la liste obtenue.

```
let codage n =
   let rec aux m =
      match m with
      |0 -> []
      |m ->   (m mod 2) :: (aux (m/2))
   in List.rev (aux n);;
```

On peut aussi utiliser une récursivité terminale.

```
let codage n =
   let rec aux reste fait =
      match m with
      |0 -> fait
      |m ->   aux (m/2) ((m mod 2) :: fait)
   in aux n;;
```

III Réalisation à base d'arbres binaires

III.1 Arbre binaire étiqueté

III.2 Arbre binaire de blocs

Question 7

Je choisis de lire supérieure en strictement supérieure.

$$AA(a) \iff \begin{cases} a \in \mathcal{F}_L \text{ ou} \\ a \in \mathcal{F}_L \text{ et } \mathcal{O}(a) > \frac{1}{2}\mathcal{T}(a) \text{ ou} \\ a \in \mathcal{N} \text{ avec } g = \mathcal{G}(a),\ d = \mathcal{D}(a) \text{ et } \begin{bmatrix} AA(g) \text{ et } AA(d) \\ \text{et non}\big(g \in\in \mathcal{F}_L \text{ et } d \in \mathcal{F}_L\big) \\ \text{et } \mathcal{T}(g) = \mathcal{T}(d) \\ \text{et } \mathcal{B}(g) = \mathcal{B}(a) + \mathcal{T}(d) \end{bmatrix} \end{cases}$$

Question 8

On va faire une démonstration par induction structurelle.
Soit $\mathcal{H}(a)$ la propriété à démontrer pour un arbre a.

- Si a est une feuille (libre ou occupé), son parcours gauche-droite ne contient qu'une feuille donc $\mathcal{H}(a)$ est vérifiée, faute de voisin de droite.
- On suppose que $\mathcal{H}(a)$ et $\mathcal{H}(a)$ sont vérifiées et a est le nœud tel que $\mathcal{G}(a) = g$ et $\mathcal{D}(a) = d$.
  $\langle f_1, \dots, f_m \rangle$ est le parcours gauche-droite de g et
  $\langle f_{m+1,\dots,f_{m+n}} \rangle$ est le parcours gauche-droite de d.
  Alors $\langle f_1, \dots, f_m, f_{m+1,\dots,f_{m+n}} \rangle$ est le parcours gauche-droite de a.
  D'après l'hypothèse, chaque feuille est adjacente à sa voisine de droite sauf f_m.
  Or on a $\mathcal{B}(f_{m+1}) = \mathcal{B}(d) = \mathcal{B}(a) + \mathcal{T}(g) = \mathcal{B}(f_1) + \mathcal{T}(g) = \mathcal{B}(f_1) + \sum_{k=1}^{m} \mathcal{T}(f_k)$
  L'adjacence des première feuilles donne $\mathcal{B}(f_{k+1}) = \mathcal{B}(f_k) + \mathcal{T}(f_k)$ pour $k < m$ donc, par une récurrence sur k, $\mathcal{B}(f_1) + \sum_{k=1}^{m-1} \mathcal{T}(f_k) = \mathcal{B}(f_m)$.
  On a ainsi $\mathcal{B}(f_{m+1}) = \mathcal{B}(f_m) + \mathcal{T}(f_m)$: la dernière adjacence est prouvée.

Question 9

Ici, il est plus direct de procéder par récurrence sur la profondeur de a'.

- Si a' est de profondeur 0 on a bien $\mathcal{T}(a) = \mathcal{T}(a') = \mathcal{T}(a').2^0$.
- On suppose que $\mathcal{T}(a) = \mathcal{T}(a').2^p$ pour tout arbre a' de profondeur p.
  Soit a'' de profondeur $p+1$, son père, a', est de profondeur p donc $\mathcal{T}(a) = \mathcal{T}(a').2^p$.
  l'autre fils de a' est de même taille que a'' donc $\mathcal{T}(a') = 2\mathcal{T}(a'')$. On aboutit alors à
  $\mathcal{T}(a) = \mathcal{T}(a').2^p = 2\mathcal{T}(a'').2^p = \mathcal{T}(a'').2^{p+1}$, la propriété est vraie au rang $p+1$.

La propriété est vraie pour toute profondeur.

Question 10

D'après le préambule du paragraphe 3, la taille d'une feuille libre indivisible est nécessairement impaire. Pour chaque feuille libre indivisible d'un arbre a on a donc $\mathcal{T}(a) = \mathcal{T}(a').2^p$ avec $\mathcal{T}(a')$ impair, or on a vu à la question **4** qu'une telle décomposition était unique.
Ainsi toutes les feuilles libres indivisibles ont la même taille p_0 et la même profondeur k_0 avec p_0 et k_0 définis par la décomposition unique $\mathcal{T}(a) = p_0.2^{k_0}$, p_0 impair.

Question 11

Il est plus simple de démontrer une généralisation :
pour tout **sous-arbre** b de a, $\mathcal{B}(b) = \mathcal{B}(a) + n_b.\mathcal{T}(b)$ où n_b est l'entier associé au chemin vers b.
On démontre alors le résultat par récurrence sur la profondeur.

- Le sous arbre de profondeur 0 est a, le chemin de a est vide, il correspond à 0 et la base de a est bien égale à la base de a augmentée de 0 fois la taille de a.
- On suppose que, pour tout sous-arbre b de a de profondeur p, $\mathcal{B}(b) = \mathcal{B}(a) + n_b.\mathcal{T}(b)$ où n_b est l'entier associé au chemin vers b.

  c est un sous-arbre de a de profondeur $p+1$.
  On note b son père. On a déjà vu que $\mathcal{T}(b) = 2\mathcal{T}(c)$.

  — Si c est le fils gauche alors on ajoute l'étiquette 0 à la fin du chemin vers b pour obtenir le chemin vers c donc $n_c = 2n_b$. De plus on a $\mathcal{B}(b) = \mathcal{B}(c)$ donc
  $\mathcal{B}(c) = \mathcal{B}(b) = \mathcal{B}(a) + n_b.\mathcal{T}(b) = \mathcal{B}(a) + n_b.2.\mathcal{T}(c) = \mathcal{B}(a) + n_c.\mathcal{T}(c)$

  — Si c est le fils droit alors on ajoute l'étiquette 1 à la fin du chemin vers b pour obtenir le chemin vers c donc $n_c = 2n_b + 1$. De plus on a $\mathcal{B}(c) = \mathcal{B}(b) + \mathcal{T}(c)$ car les deux fils ont la même taille donc
  $\mathcal{B}(c) = \mathcal{B}(b)+\mathcal{T}(c) = \mathcal{B}(a)+n_b.\mathcal{T}(b)+\mathcal{T}(c) = \mathcal{B}(a)+n_b.2.\mathcal{T}(c)+\mathcal{T}(c) = \mathcal{B}(a)+n_c.\mathcal{T}(c)$.

  — Dans les deux cas on trouve la formule recherchée.

- On a prouvé le résultat par récurrence.

III.3 Opérations

Question 12

```
let rec maximum a =
   match a with
   |Libre t -> t
   |Occupe(occ, tot) -> 0
   |Noeud(g, d) -> max (maximum g) (maximum d);;
```

Question 13

Les contraintes d'adjacences ont été intégrées dans le type car il n'y a plus de base.
Pour tenir compte des contraintes d'égalité de taille, on va créer une fonction auxiliaire qui retourne un couple formé du booléen de vérification et de la taille de l'arbre.

```
let verifier arbre =
  let rec aux a =
    match a with
      |Libre t -> true, t
      |Occupe(occ, tot) -> (occ > tot/2), tot
      |Noeud(Libre tg, Libre td) -> false, (tg + td)
      |Noeud(g, d) -> let (bg, tg) = aux g in
                      let (bd, td) = aux d in
                      (bg && bd & (tg = td) , tg + td)
  in fst (aux arbre);;
```

Question 14

Lorsqu'on libère le bloc de base 2 il devient libre et on obtient un arbre avec deux fils qui sont des feuilles libres. Comme cela est interdit par la condition **C1**, on fusionne les deux blocs en le bloc libre de paramètres 2 et 2.
On est alors de nouveau en conflit avec la condition **C1** et il faut fusionner de nouveau.
On aboutit à `Noeud(Libre 4, Occupé(3, 4))`.

Question 15

```
let rec liberer c a =
  match c, a with
     |[], Occupe(occ, tot) -> Libre tot
     |0::q, Noeud(g, Libre td) -> begin match liberer q g with
                                   |Libre tg -> Libre (tg + td)
                                   |lg -> Noeud (lg, Libre td) end
     |1::q, Noeud(Libre tg, d) -> begin match liberer q d with
                                   |Libre td -> Libre (tg + td)
                                   |ld -> Noeud (Libre tg, ld) end
     |0::q, Noeud(g, d) -> Noeud (liberer q g, d)
     |1::q, Noeud(g, d) -> Noeud (g, liberer q d)
     |_ -> failwith "Opération non conforme";;
```

Question 16

Le meilleur cas est celui de l'arbre constitué d'un unique bloc alloué à libérer. On n'a aucun appel récursif et la complexité est constante.
Le pire des cas est celui où le bloc à libérer est de profondeur maximale, la complexité est alors linéaire en la profondeur.

Question 17

On ne peut allouer une taille 1 que dans un bloc de taille 1 en raison de la condition **C4**.
On découpe le bloc libre en 2 puis un de ses fils en 2 aussi.
On aboutit à `Noeud(Noeud(Ocuppe(1, 1), Libre 2), Libre 4)`.

Question 18

Comme on veut allouer dans le plus petit bloc libre possible on doit calculer le chemin vers un bloc de taille suffisante mais le plus petit possible. On commence par une première fonction qui fait ce calcul, le choix dépend de la taille de chaque coté donc la fonction renvoie aussi la taille trouvée. La taille trouvée est 0 s'il n'y a pas de bloc assez grand.

```
let rec chemin arbre k =
   match arbre with
   |Libre t when t >= t -> ([], t)
   |Libre t -> ([], 0)
   |Occupe(occ, tot) -> ([], 0)
   |Noeud(g, d) -> let (chemg, tg) = chemin g k in
                   let (chemd, td) = chemin d k in
                   if tg = 0 || td < tg
                   then (1::chemd, td)
                   else (0::chemg, tg);;
```

On écrit ensuite la fonction de découpage d'un bloc libre, elle renvoie aussi le chemin suivi, celui-ci ne contient que des 0 car on choisit d'allouer à gauche..

```
let rec decoupage arbre k =
   match arbre with
   |Libre t when t < 2*k -> [], Occupe (k, t)
   |Libre t -> let chem, g = decoupage (Libre (t/2)) k in
               0::chem, Noeud(g, Libre (t/2))
   |_ -> failwith "Opération non conforme";;
```

On suit alors le chemin pour trouver le bloc libre et, éventuellement, le découper.

```
let allouer k arbre =
   let chem, taille = chemin arbre k in
   if taille = 0
   then ([-1], arbre)
   else let rec alloc c a =
           match c, a with
           |[], Libre t -> decoupage a k
           |0::q, Noeud(g, d) -> let ca, ga = alloc q g in
                                 0::ca, Noeud(ga, d)
           |1::q, Noeud(g, d) -> let ca, da = alloc q d in
                                 1::ca, Noeud(g, da)
           |_ -> failwith "Opération non conforme"
        in alloc chem arbre;;
```

Question 19

Le meilleur cas est celui de l'arbre a constitué d'un unique bloc libre avec $\frac{1}{2}\mathcal{T}(a) < t \leq \mathcal{T}(a)2$: il n'y a alors aucun appel récursif.
Notons que la fonction `chemin` doit parcourir tout les nœuds de l'arbre. La complexité est alors linéaire en la taille de l'arbre, les autres fonctions ne parcourent qu'une branche.

Question 20

?????