DS4 d'informatique MPSI
10 juin 2011

Devoir surveillé

Durée : 2 heures

Aucun document n'est autorisé. **On pourra définir des fonctions non demandées explicitement, si cela facilite la programmation. Inutile de prouver la correction et la terminaison des fonctions écrites, sauf si on le demande explicitement. Les fautes de syntaxe seront sanctionnées.**

Listes à deux bouts de nombres dyadiques

L'énoncé utilise à plusieurs reprises la formulation « on garantira l'invariant P sur le type τ ». On entend par cela que les fonctions que vous allez écrire peuvent supposer que la propriété P est vraie pour leurs arguments de type τ et qu'en contrepartie elles doivent produire des résultats de type τ vérifiant la propriété P.

Partie A – Grands entiers

Les nombres que nous allons manipuler nécessitent une précision qui dépasse celle des entiers de la machine (type `int`). Nous allons donc commencer par définir une arithmétique de précision arbitraire. On se donne pour cela une base de calcul, par exemple `base`= 10000. La valeur de `base` importe peu et on supposera seulement qu'elle est paire, supérieure ou égale à 2 et que son double n'excède pas le plus grand entier machine. Un entier naturel de précision arbitraire est alors représenté par la liste de ses chiffres en base `base`, les chiffres les moins significatifs étant en tête de liste. Ainsi la liste [1; 2; 3] représente l'entier $1 + 2 \times \texttt{base} + 3 \times \texttt{base}^2$. On définit le type `nat` suivant pour de tels entiers :

```
let base = ... ;;
type nat == int list ;;
```

Dans la suite, on garantira l'invariant suivant sur le type `nat` :
- tout élément de la liste est compris entre 0 et `base`-1, au sens large ;
- le dernier élément de la liste, lorsqu'il existe, n'est pas nul.

On notera que l'entier 0 est représenté par la liste vide.

A.1 Définir une fonction `cons_nat` qui prend en argument un chiffre c (un entier machine, $0 \leqslant c <$ `base`) et un grand entier n et qui renvoie le grand entier $c + \texttt{base} \times n$. La fonction `cons_nat` peut aider à garantir l'invariant du type nat

```
cons_nat : int -> nat -> nat
```

A.2 Définir une fonction `cmp_nat` qui prend en argument deux grands entiers n_1 et n_2 et qui renvoie un entier machine valant, -1 si $n_1 < n_2$, 1 si $n_1 > n_2$ et 0 si $n_1 = n_2$.

```
cmp_nat : nat -> nat -> int
```

A.3 définir une fonction `add_nat` qui calcule la somme de deux grands entiers. Indication : on pourra commencer par écrire une fonction prenant également une retenue en argument et appliquer l'algorithme traditionnel enseigné à l'école primaire.

```
add_nat : nat -> nat -> nat
```

A.4 Définir une fonction `div2_nat` qui prend en argument un grand entier n et qui calcule le quotient et le le reste dans la division euclidienne de n par 2. Le quotient est un grand entier et le reste un entier machine valant 0 ou 1. On rappelle que la constante `base` est paire.

```
div2_nat : nat -> nat * int
```

À partir de ces grands entiers naturels, on va maintenant construire de grands entiers relatifs. Pour cela, on introduit le type enregistrement `z` suivant, où le champ `signe` contient le signe de l'entier relatif, à savoir 1 ou -1, et le champ `nat` sa valeur absolue.

```
type z = { signe: int; nat: nat };;
```

On notera que 0 admet deux représentations, ce qui n'est pas gênant pour la suite.

A.5 Définir une fonction `neg_z` qui calcule la négation d'un grand entier relatif.

```
neg_z : z -> z
```

A.6 Définir une fonction `add_z` qui calcule la somme de deux entiers relatif.

```
add_z : z -> z -> z
```

A.7 Définir une fonction `mul_puiss2_z` qui prend en arguments un entier machine p ($p \geqslant 0$), un grand entier relatif z, et qui renvoie le grand entier relatif $2^p z$. On se contentera d'une solution simple, sans viser particulièrement l'efficacité.

```
mul_puiss2_z : int -> z -> z
```

A.8 Définir une fonction `decomp_puiss2_z` qui prend en argument un grand entier relatif z non nul, et qui renvoie un grand entier relatif u impair et un entier machine p tels que $z = 2^p u$. Cette fonction calcule donc la plus grande puissance de 2 qui divise z et renvoie la décomposition correspondante. Comme ci-dessus, on visera la simplicité et on supposera que z est tel que p est bien représentable par un entier machine.

```
decomp_puiss2_z : z -> z * int
```

Partie B – Nombres dyadiques

Un nombre dyadique est un nombre rationnel qui peut s'écrire sous la forme

$$a \times 2^b \quad \text{avec} \quad a, b \in \mathbb{Z}$$

On note $\mathcal{D}$ l'ensemble des nombres dyadiques. On définit le type `dya` suivant pour représenter les nombres dyadiques :

```
type dya = { m : z ; e : int } ;;
```

Si d est une valeur de type `dia`, on l'interprète donc comme le nombre rationnel $d.\mathtt{m} \times 2^{d.\mathtt{e}}$, où $d.\mathtt{m}$ est parfois appelé *mantisse* et $d.\mathtt{e}$ *exposant*.

On garantira l'invariant suivant sur le type `dya` : la valeur du champ `m` est soit nulle soit impaire. On supposera par ailleurs que la capacité des entiers machines ne sera jamais dépassée dans le calcul des exposants.

B.1 Définir une fonction `div2_dya` qui divise un nombre dyadique par 2.

```
div2_dya : dya -> dya
```

B.2 Définir une fonction `add_dya` qui calcule la somme de deux nombres dyadiques.

```
add_dya : dya -> dya -> dya
```

B.3 Définir une fonction `sous_dya` qui calcule la différence de deux nombres dyadiques.

```
add_dya : dya -> dya -> dya
```

Partie C – Listes à deux bouts

On considère maintenant des listes de nombres dyadiques. Si une telle liste contient les n éléments $x_1, x_2, \ldots, x_n$ dans cet ordre, on la note $\langle x_1; x_2; \ldots; x_n \rangle$. Nous cherchons à manipuler de telles listes aux deux extrémités, c'est à dire d'ajouter et de supprimer des éléments à gauche comme à droite, et également de calculer efficacement l'image miroir d'une telle liste, c'est à dire la liste $\langle x_n; \ldots; x_2; x_1 \rangle$. La notion usuelle de liste se prêtant mal à de telles opérations (seule la manipulation de l'extrémité gauche de la liste est aisée), l'objectif de cette partie est de réaliser une structure de données raisonnablement efficace pour représenter une telle « liste à deux bouts ». Pour éviter les confusions, nous utiliserons désormais le terme de "LDB" pour désigner une liste à deux bouts, et nous continuerons d'utiliser le terme de « liste » pour désigner une liste usuelle.

L'idée est d'utiliser non pas une liste mais deux pour représenter une LDB, la première liste représentant la partie gauche de la LDB et la seconde liste sa partie droite. Ainsi, l'ensemble des deux listes g = [1; 2] et d = [5; 4; 3] représentera la LDB < 1; 2; 3; 4; 5>. La liste g contient les premiers éléments de la LDB, dans le bon ordre, et la tête de cette liste coïncide donc avec l'extrémité gauche de la LDB ; symétriquement, la liste d contient les derniers éléments de la LDB, en ordre inverse, et la tête de cette liste coïncide donc avec l'extrémité droite de la LDB.

On définit le type `ldb` suivant pour représenter les LDB :

```
type ldb = {
lg : int ; g : dya list ;
ld : int ; d : dya list  };;
```

On se donne une constante entière $c \geqslant 2$ et on impose sur le type `ldb` les deux invariants suivants :

(1) le champ `lg` contient la longueur de la liste `g`, et le champ `ld` celle de la liste `d`

(2) `lg` $\leqslant c \times$ `ld` $+ 1$ et `ld` $\leqslant c \times$ `lg` $+ 1$.

Toutes les questions de cette partie garantiront les invariants au sens précisé dans l'introduction du problème.

C.1 Définir une fonction `ldb_est_vide` qui détermine si une LDB est vide.

```
ldb_est_vide : ldb -> bool
```

C.2 Définir une fonction `premier_g` qui renvoie l'élément le plus à gauche d'une LDB, *i.e.* telle que `premier_g`$\langle x_1; x_2; \ldots; x_n \rangle = x_1$. On supposera que la LDB contient au moins un élément.

```
premier_g : ldb -> dya
```

C.3 Définir une fonction `inverse_ldb` qui inverse l'ordre des éléments d'une LDB, *i.e.* telle que `inverse_ldb`$\langle x_1; x_2; \ldots; x_n \rangle = \langle x_n; \ldots; x_2; x_1 \rangle$.

```
inverse_ldb : ldb -> ldb
```

On suppose désormais disposer d'une fonction `invariant_ldb` qui vérifie si une LDB satisfait bien l'invariant (2) et le rétablit si ce n'est pas le cas. Plus précisément, la fonction `invariant_ldb` renvoie son argument inchangé lorsqu'il vérifie l'invariant et, dans le cas contraire, renvoie une LDB de même contenu vérifiant l'invariant.

```
invariant_ldb : ldb -> ldb
```

C.4 Définir une fonction `ajoute_g` qui ajoute un élément à gauche d'une LDB, *i.e.* telle que `ajoute_g` x $\langle x_1; x_2; \ldots; x_n \rangle = \langle x; x_1; x_2; \ldots; x_n \rangle$.

```
ajoute_g : dya -> ldb -> ldb
```

C.5 Définir une fonction `enleve_g` qui supprime l'élément le plus à gauche dans une LDB, *i.e.* telle que `enleve_g` $\langle x_1; x_2; \ldots; x_n \rangle = \langle x_2; \ldots; x_n \rangle$. On supposera que la LDB contient au moins un élément.

```
enleve_g : ldb -> ldb
```

C.6 Dans cette question, on suppose $c = 3$. On considère une LDB de longueur N obtenue en appliquant successivement N opérations `ajoute_g` à partir d'une LDB vide. Quel est le coût *moyen* de chaque opération `ajoute_g` ? On supposera que le coût de l'opération `invariant_ldb` est constant lorsque la LDB vérifie l'invariant (2) et proportionnel à la longueur de la LDB lorsque celle-ci est réarrangée.