Typage avancé

Objectifs

— Utiliser des notions de typage avancé : types fantômes, GADT, types extensibles.
— Listes dépendantes. Assembleur typé.

On s'intéresse à l'introduction de paramètres supplémentaires dans le type `list`, afin de spécifier plus précisément le comportement de certaines fonctions. On aura besoin des types `zero` et `_ succ` pour représenter des entiers naturels, du type `nil` pour indiquer la fin d'une liste, et enfin du type `'a list` :

```
type zero = private Dummy1
type _ succ = private Dummy2
type nil = private Dummy3
type 'a list = Nil | Cons of 'a * 'a list
```

1 n-listes

On s'intéresse d'abord aux n-listes, où l'on spécifie dans le type la taille de la liste.

▷ **Exercice 1 (n-listes)**

1. *Définir le type `type ('a, 'n) nlist` représentant les listes d'éléments de type `'a` et de taille `'n`. On reprendra les mêmes contructeurs `Nil` et `Cons`.*
2. *Définir la fonction `map : ('a -> 'b) -> ('a, 'n) nlist -> ('b, 'n) nlist` sur le modèle de `List.map`.*
3. *Donner le type et définir la fonction `snoc` qui ajoute un élément à la fin d'une n-liste.*
4. *Donner le type et définir la fonction `tail` qui prend une n-liste non vide et renvoie la queue de cette n-liste. Cette fonction ne doit pas lever d'erreurs à l'exécution.*
5. *Définir la fonction `rev : ('a, 'n) nlist -> ('a, 'n) nlist` sur le modèle de `List.rev`.*

On va maintenant spécifier un algorithme plus complexe : le tri par insertion, dont une version fonctionnelle classique est donnée ci-dessous. On prouve que ce tri préserve la longueur de la liste.

```
let rec insert x l =
  match l with
  | []   -> x::l
  | t::q -> if t < x then t::insert x q else x::l;;
let rec insertion_sort l =
  match l with
  | []   -> []
  | t::q -> insert t (insertion_sort q);;
```

▷ **Exercice 2 (Tri par insertion)**
Transformer l'algorithme de tri par insertion donné ci-dessus afin qu'il manipule des `('a, 'n) nlist` et que les annotations de type prouvent la propriété de préservation de la longueur de la liste.

2 h-listes

On s'intéresse maintenant aux listes hétérogènes, ou h-listes. Une h-liste accepte des éléments de tout type. Le paramètre de type représente la suite des types (potentiellement différents) de tous les éléments de la h-liste. Cette suite est écrite sous la forme d'un produit de types, terminé par `nil`. Par exemple, on aurait le typage suivant : `Cons (1, Cons (true, Nil)) : (int * (bool * nil)) hlist`

▷ **Exercice 3 (Les h-listes)**

1. *Définir le type `'p hlist` représentant les listes hétérogènes.*
2. *Donner le type et définir la fonction `tail` qui prend une h-liste (non vide) et renvoie la queue de cette h-liste. Cette fonction ne doit pas lever d'erreurs à l'exécution.*
3. *Donner le type et définir la fonction `add` qui prend deux entiers aux deux premières positions d'une h-liste et insère leur somme en tête de la queue (la h-liste privée des deux premiers entiers). Cette fonction ne doit pas lever d'erreurs à l'exécution.*

3 Assembleur typé

On s'intéresse à l'évaluation et à la compilation du langage d'expressions suivant, qui permet de décrire des valeurs entières `Entier` v ou booléennes `Booleen` v, des additions `Plus (`e_1`,` e_2`)` (valides entre entiers uniquement) et des égalités `Egal (`e_1`,` e_2`)`.

```
type 't expr = Entier : int -> int expr | Booleen : bool -> bool expr
             | Plus : int expr * int expr -> int expr | Egal : 't expr * 't expr -> bool expr
```

▷ **Exercice 4 (Évaluation d'expressions bien typées)**
Définir une fonction d'évaluation `eval : 't expr -> 't` calculant la valeur d'une expression.

Le langage cible de la compilation du langage d'expressions est un assembleur pour une machine à pile, contenant les instructions suivantes : `PushI` v et `PushB` v qui empilent une valeur v (respectivement entière ou booléenne) ainsi que `Add` et `Equ` qui dépilent 2 arguments en sommet de pile et empilent le résultat de l'opération (respectivement addition entre entiers ou test d'égalité entre valeurs du même type). On peut bien sûr écrire des séquences d'instructions avec le constructeur `Seq (`i_1`,` i_2`)`. Par exemple, la compilation de l'expression `Plus (Entier 1, Entier 2)` pourra produire le code suivant : `Seq(PushI 1, Seq(PushI 2, Add))`. L'exécution de ce code procède alors ainsi : l'exécution de `PushI 1` commence par empiler 1, puis on empile 2, et enfin l'exécution de `Add` dépile les deux entiers en sommet de pile et empile leur somme, soit 3. On obtient donc la valeur de l'expression initiale en sommet de pile. On propose les types `valeur` et `code` suivants :

```
type valeur = Int of int | Bool of bool
type code = PushI of int | PushB of bool | Add | Equ | Seq of code * code
```

▷ **Exercice 5 (Compilation et exécution)**

1. *Définir la fonction `compile : 't expr -> code` qui produit un code à partir d'une expression.*
2. *Définir la fonction `exec : code -> valeur list -> valeur list`, qui prend un code à exécuter, une pile de valeurs initiale donnée sous forme de liste et produit une nouvelle pile de valeurs.*

On constate que l'écriture de la fonction `exec` est alourdie par le traitement des erreurs. Ce phénomène serait très accentué si l'on voulait traiter un langage d'expressions et un assembleur plus complets. Pour éviter ce problème, sur le même principe que pour les expressions, on se propose de définir un assembleur typé en ajoutant au type `code` et à ses constructeurs deux paramètres supplémentaires, pour représenter précisément leur sémantique de transformation de piles.

▷ **Exercice 6 (Assembleur typé)**
Définir le nouveau type `('stin, 'stout) code`*, où les paramètres* `'stin` *et* `'stout` *représentent respectivement l'état de la pile avant et après l'exécution du code. On s'inspirera du type* `_ hlist`*.*

On veut enfin redéfinir `compile` et `exec` en fonction des nouveaux types introduits, sachant que les piles sont représentées par des h-listes.

▷ **Exercice 7 (Compilation et exécution typée)**
Donner les types et redéfinir les nouvelles fonctions `compile` *et* `exec`*. Aucun cas d'erreur ne doit subsister dans cette dernière.*