TP 3 de programmation fonctionnelle en Objective Caml

Christian Rinderknecht

4 février 2015

L'objectif est de présenter un usage fréquent en pratique des exceptions, à travers la fonction prédéfinie `failwith`. La seconde partie illustre l'utilité des types polymorphes récursifs grâce au type prédéfini `list`. Il s'agit de comprendre aussi que ce type est seulement un cas particulier, dédié à un usage précis, malgré la syntaxe spéciale de ses constructeurs.

1 `failwith`

Jusqu'ici, les fonctions que nous avons écrites renvoyaient toujours un résultat. Entre autres conséquences, toute construction `if` devait comporter un `else`, de façon à ce que le résultat soit défini, indépendamment de la valeur de la condition.

Cela était parfois gênant. Par exemple, lorsque nous avons écrit une fonction qui renvoie les deux racines d'un polynôme du second degré, vérifier que le discriminant était positif ne servait à rien, puisque dans le cas contraire, nous devions quand même renvoyer un résultat—nous n'avions aucun moyen de signaler l'erreur.

Pour remédier à cette situation, il existe une fonction prédéfinie en Caml, nommée `failwith`, définie par `let failwith x = raise (Failure x)` et a donc le type $\forall\alpha.\text{string} \to \alpha$. Elle prend en argument une chaîne de caractères. Son effet est d'arrêter immédiatement l'exécution du programme et d'afficher la chaîne de caractères qu'elle a reçue. `failwith` ne renvoie pas de résultat, puisqu'elle arrête le programme ; du point de vue du typage, son résultat peut donc prendre n'importe quel type. C'est pourquoi on donne à celui-ci le type α, sans spécifier la valeur de α.

Par exemple, considérons l'expression

```
if delta >= 0.0
then ((-.b +. sqrt(delta)) /. (2. *. a),
      (-.b -. sqrt(delta)) /. (2. *. a))
else failwith "Erreur: discriminant négatif"
```

La première clause du `if` a le type `float` $\times$ `float`. La seconde a le type α, comme expliqué ci-dessus. Rappelons que les deux clauses d'un `if` doivent avoir le même type. Il en découle que $\alpha =$ `float` $\times$ `float`, et que l'expression complète est bien typée.

Pour conclure, on peut utiliser `failwith` lorsqu'on écrit une fonction dont le résultat n'est pas défini dans certains cas. Nous allons en voir quelques exemples ci-dessous.

2 Listes

Syntaxe : La liste vide s'écrit `[]`. On peut construire une liste en donnant tous ses éléments entre crochets, séparés par des points-virgules, par exemple : `[ 1; 5; x ]`. La tête de liste est l'élément le plus à gauche. Pour ajouter un élément `x` en tête d'une liste `l`, on écrit `x :: l`. On remarque que `[ 1; 5; x ]` n'est en fait qu'une abréviation pour `1 :: 5 :: x :: []`.

Les motifs servant à filtrer des listes ont la même syntaxe. `[]` filtre la liste vide, `x :: l` filtre une liste à au moins un élément, et `[ m1; ... ; mn ]` filtre une liste à exactement n éléments.

L'ensemble α `list` des listes d'éléments de type α peut être décrit par l'équation

$$\alpha \texttt{ list} = \{\texttt{[]}\} \cup (\alpha \texttt{ :: } \alpha \texttt{ list})$$

Pour comprendre cette équation, rappelons qu'on peut définir l'ensemble $\mathbb{N}$ des entiers par

$$\mathbb{N} = \{0\} \cup (1 + \mathbb{N})$$

L'analogie entre ces deux équations signifie que l'on peut raisonner par récurrence sur les listes, de façon similaire à ce que l'on fait sur les entiers. Un raisonnement (ou une fonction) par récurrence sur les entiers doit traiter deux cas : le cas de base (en général $n = 0$) et le cas général (passage de n à $n + 1$). Dans le cas général, on utilise une hypothèse de récurrence (s'il s'agit d'une preuve) ou un appel récursif (s'il s'agit d'une fonction). De même, un raisonnement (ou une fonction) par récurrence sur les listes traitera le cas de base (la liste vide `[]`) et le cas général (passage de `l` à `x :: l`). Dans le cas général, on utilisera aussi un appel récursif pour traiter la sous-liste `l`.

Nous allons écrire quelques fonctions classiques de manipulation de listes.

- Écrire une fonction `appartient`, qui prend en argument un élément `x` et une liste `l`, et renvoie un booléen indiquant si `x` figure ou non dans `l`.
- Plus généralement, écrire une fonction `existe`, qui prend en argument un prédicat `p` et une liste `l`, et renvoie un booléen indiquant si il existe un élément de `l` qui vérifie `p`. Application : réécrire `appartient` en utilisant `existe`.
- Ecrire une fonction `associé` définie comme suit. `associé` prend en argument un élément `x` et une liste de paires `l`. Elle renvoie la deuxième composante du premier élément de la liste dont la première composante est `x`. Par exemple, `associé "y" [("x", 1); ("y", 2); ("z", -4)]` renvoie 2. (Si aucune paire dans `l` ne commence par `x`, on utilisera `failwith` pour signaler l'erreur.)
- Écrire une fonction `map` qui prend en argument une fonction `f` et une liste `l`, et renvoie la liste des images des éléments de `l` par `f`, i.e. `map f [ x1; ...; xn ]` doit être égal à `[ f(x1); ...; f(xn) ]`.
- Écrire une fonction `split` qui prend en argument une liste de couples et les sépare pour renvoyer un couple de listes.

— Écrire une fonction `partage` qui prend en argument un prédicat `p` et une liste `l` et renvoie un couple formé de la liste des éléments de `l` qui vérifient `p` et de la liste de ceux qui ne le vérifient pas.
— Écrire une fonction `append`, qui prend deux listes en argument et les concatène.
— On veut écrire une fonction `fold_right` qui prend en argument une fonction `f`, une liste `[ a1; ...; an ]` et une valeur `x` et calcule

$$f\, a_1\, (f\, a_2\, (\ldots (f\, a_{n-1}\, (f\, a_n\, x)) \ldots))$$

Quel sera le type de `fold_right` ? Exprimer `fold_right f (a1 :: l) x` en fonction de `fold_right f l x`. En déduire comment écrire la fonction.
— Mêmes questions pour une fonction `fold_left` qui prend en argument une fonction `f`, une valeur `x` et une liste `[ a1; ...; an ]` et calcule

$$f\, (f\, (\ldots (f\, (f\, x\, a_1)\, a_2) \ldots)\, a_{n-1})\, a_n$$

— Réécrire la fonction `append` en utilisant `fold_right`.

3 Tri par fusion (*Merge sort*)

Dans cette section, on se propose de programmer un algorithme de tri sur les listes, dit *tri par fusion*.

L'idée est de commencer par programmer l'opération de fusion, qui prend deux listes déjà triées et en fait une seule liste triée.

Ensuite, on transforme notre liste à trier, contenant n éléments, en n listes à un élément. Puis on fusionne ces listes deux par deux, ce qui nous donne $n/2$ listes triées à deux éléments, plus éventuellement une liste à un élément (si n est impair). On les fusionne à nouveau deux par deux, etc. jusqu'à ce qu'il ne reste plus qu'une seule liste, qui est alors triée.

Il est intéressant de noter que le type des éléments manipulés n'a pas d'importance, parce que la fonction de comparaison `<` est polymorphe : elle a le type $\forall\alpha.\alpha \to \alpha \to$ `bool`. Cela permet d'utiliser la même fonction `mergesort` pour trier des listes d'entiers, de flottants, de listes, etc. sans limitation.

— Écrire une fonction qui prend en argument une liste `[ x1; ...; xn ]` et renvoie la liste des listes à un élément `[ [x1]; ...; [xn] ]`. On utilisera `map`.
— Écrire une fonction `merge` qui prend en argument deux listes triées par ordre croissant et les fusionne en une seule liste triée.
— Écrire une fonction `merge2à2` qui prend en argument une liste de listes `[ l1; l2; l3; l4; ... ]` et renvoie une liste où les listes voisines ont été fusionnées, i.e. `[ merge l1 l2; merge l3 l4; ... ]`. On prendra garde à traiter correctement le cas où la liste d'entrée est de longueur impaire.
— En combinant les fonctions précédentes, écrire une fonction `mergesort` qui prend une liste en argument et la trie. Pour cela, on crée la liste des listes à un élément, puis on lui applique `merge2à2` itérativement, jusqu'à obtenir une liste de la forme `[ l ]`. On renvoie alors `l`.

— Exprimer le nombre maximal de comparaisons nécessité par chacune des fonctions ci-dessus, en fonction de la taille de son argument. En déduire que le tri d'une liste de taille n demande un temps $O(n \cdot \log_2 n)$.

Les questions précédentes ont été réalisées en utilisant la fonction de comparaison générique `<`. Les réécrire en utilisant une fonction d'ordre `ordre` que l'on passera en argument partout où c'est nécessaire. Utiliser ces nouvelles fonctions pour trier une liste par ordre décroissant.

4 Tri rapide (*Quicksort*)

Voici maintenant un autre algorithme de tri. L'algorithme procède comme suit.

On choisit un élément `p` de la liste, appelé *pivot*. On sépare ensuite la liste en trois parties : le pivot `p`, la liste `l1` des éléments plus petits que `p`, et la liste `l2` des éléments plus grands que, ou égaux à `p`. (On peut utiliser la fonction `partage` écrite précédemment.) La liste triée est alors égale à la concaténation de `l1` triée, `p` et `l2` triée.