Correction du TD 3

Julien Reichert

Dichotomie pour la recherche dans un tableau trié

On écrit ici une fonction qui localise une occurrence de l'élément recherché. Si l'élément n'est pas dans le tableau, la fonction déclenche l'exception `Not_found`. Si le tableau n'est pas croissant, le comportement est arbitraire.

```
let dichotomie t x =
  let rec aux deb fin =
    if deb > fin then raise Not_found;
    let mil = (deb + fin) / 2 in
    if t.(mil) = x then mil
    else if t.(mil) < x then aux (mil + 1) fin
    else aux deb (mil - 1)
  in aux 0 (Array.length t - 1);;
```

La preuve de terminaison repose sur la valeur `fin - deb + 1`, qui est un entier naturel (la récursion s'arrête s'il n'est plus strictement positif) qui décroît strictement à chaque appel récursif imbriqué. Comme en pratique il est divisé par deux à chaque étape (`fin - deb` est divisé par deux avec arrondi inférieur ou supérieur suivant le cas, et on ôte encore une unité), on peut aussi considérer la partie entière se son logarithme en base deux (le `+ 1` servait à ne pas calculer le logarithme en base deux de zéro), qui décroît aussi strictement à chaque étape et permet de déduire que la complexité est logarithmique en la taille du tableau, qui est à une unité près la valeur initiale du variant, puisqu'un appel récursif ne consiste qu'à faire des opérations de coût constant et l'appel suivant.

La preuve de correction repose sur la propriété suivante, qu'on établit par « récurrence forte descendante »sur la valeur de `fin - deb` et que vérifie la fonction `aux` : « La valeur recherchée est dans le tableau si, et seulement si, il y est entre l'indice `deb` et l'indice `fin` ». L'initialisation se fait quand `fin - deb` vaut la taille du tableau moins un, où elle est triviale, et l'hérédité se prouve en établissant (par transitivité de la relation de comparaison), qu'un appel récursif (pour lequel on suppose l'équivalence annoncée par la propriété) permet soit de localiser l'élément, soit de relancer un appel récursif dans un intervalle plus restreint dont on a retiré les indices où l'élément cherché ne pouvait pas se trouver. Ainsi, si l'intervalle devient vide, on sait que l'élément n'est pas dans le tableau.

Le concept de récurrence forte descendante peut certes perturber, et on pourra considérer à la place une récurrence simple sur le nombre d'appels récursifs...

Dichotomie pour la recherche d'un zéro d'une fonction continue

On fournit un argument supplémentaire déterminant le seuil de précision attendu. Quand l'intervalle considéré est de taille inférieure à ce seuil, on arrête la récursion.

```
let recherche_zero f a b eps =
  if f a *. f b > 0. then failwith "Mauvais intervalle de départ";
  (* On ne va pas vérifier ceci à chaque fois, d'où la fonction aux. *)
  if f a = 0. then a else if f b = 0. then b else if a = b then a else
  let rec aux deb fin =
    let mil = (deb +. fin) /. 2. in
    if (fin -. deb) < eps || f mil = 0. then mil
    else if f deb *. f mil < 0. then aux deb mil
    else aux mil fin
  in aux a b;;
```

Pour prouver la terminaison, on considère la partie entière du logarithme en base deux de l'écart entre `fin` et `deb` dans la fonction `aux`. Il s'agit d'un entier strictement décroissant à chaque appel récursif imbriqué, et il est minoré par la partie entière du logarithme en base deux de la valeur `eps`, garantissant une profondeur de récursion limitée. La différence entre ces logarithmes, autrement dit le logarithme du rapport, domine la complexité puisque là aussi le reste de chaque appel récursif se limite à des opérations en temps constant.

La correction repose également sur la propriété selon laquelle il existe toujours un antécédent de zéro par la fonction dans l'intervalle considéré, puisque la vérification initiale permet de confirmer qu'il en existe un entre les bornes en argument.

Première fonction

Spécification : il s'agit d'un mélange pseudo-aléatoire, avec la propriété que si le générateur aléatoire suit une loi uniforme (ce qu'on ne peut pas garantir mais qu'on admettra en principe), alors toutes les permutations sont équiprobables.

La terminaison est garantie par l'absence de boucles conditionnelles et de récursions, et la complexité (en faisant abstraction de l'utilisation de l'aléa…) est linéaire en la taille du tableau car chaque tour de boucle fait appel à des opérations et des appels de fonctions de complexité constante.

Pour la preuve de correction, on notera que plus la spécification est précise, plus la preuve est difficile : si on se contente de dire qu'on mélange la liste, il suffira de remarquer que le corps de boucle ne fait que des transpositions. Ici, on pourra faire une récurrence et dire qu'en plus du fait que c'est un mélange, au premier tour de boucle l'élément envoyé à la dernière position y restera définitivement, au deuxième c'est l'avant-dernière position qui est fixe, etc. En plus de cela, on notera que chaque élément non encore figé a la même probabilité de se faire figer à l'étape en cours. Il est possible que la mise en forme de tout ceci nécessite une avancée supplémentaire en cours de mathématiques, cependant…

Deuxième fonction

Spécification : la fonction prend en argument une chaîne de caractères et retourne le miroir de cette chaîne.

La terminaison est là aussi garantie. Ceci étant, pour le calcul de la complexité, il faut se poser la question de l'unité la plus pertinente. Mesurer l'évolution du temps mis en décuplant la taille de la chaîne en argument suggère de choisir le recollement d'un caractère à une chaîne en construction ou le recopiage d'un caractère dans la mémoire (en créant une chaîne ou en mettant à jour une référence de chaînes). Pour cette raison, chaque tour de boucle a une complexité linéaire en la taille de la chaîne référencée, pour un total quadratique.

Pour la preuve de correction, l'invariant à utiliser est l'information qu'après le tour numéro `i`, les `i+1` premiers caractères de la chaîne figurent dans le sens inverse dans la référence `rep`.

Troisième fonction

Spécification : la fonction prend en argument deux chaînes de caractères dont la première sera appelée « la grosse chaîne » et la deuxième « le motif », et retourne le nombre d'indices de la grosse chaîne où commence une occurrence complète du motif, sachant que plusieurs occurrences peuvent se chevaucher.

On retrouve les difficultés de la fonction précédente : le test de comparaison dans la boucle cache une boucle comparant caractère par caractère deux chaînes, suggérant qu'une telle comparaison soit l'opération élémentaire à retenir, pour une complexité de l'ordre du produit des tailles des chaînes.

Pour la preuve de correction, le court-circuitage d'une boucle facilite grandement les choses, l'invariant sera que la variable `rep` référence le nombre de succès jusqu'à l'indice que l'on vient d'étudier. Initialisation et hérédité sont faciles à établir.

Quatrième fonction

Spécification : la fonction prend en argument une liste et détermine si elle est croissante.

Terminaison : la taille de la liste est un variant, elle décroît d'un à chaque appel récursif imbriqué.

Complexité : trivialement linéaire.

Correction : On prouve « les mathématiques derrière le programme » par récurrence sur la taille de la liste (en attendant l'induction au deuxième semestre…). Une liste de taille zéro ou un est forcément croissante, pour une taille supplémentaire une liste est croissante si, et seulement si, son premier élément est inférieur au deuxième et qu'en excluant son premier élément on ait une liste croissante.

Cinquième fonction

Spécification : la fonction prend en argument une liste d'entiers et détermine si elle correspond aux premiers termes d'une suite arithmétique.

Terminaison et complexité : comme la fonction précédente.

Correction : Comme la fonction précédente, ici le principe mathématique est que l'écart entre deux termes consécutifs (cas de récurrence quand il y en a au moins trois) est constant. On vérifie donc l'écart entre les deux premiers et celui entre le deuxième et le troisième, en bousculant l'équation dans le programme pour éviter que ce ne soit trop simple.

Sixième fonction

Spécification : la fonction prend en argument deux listes dont les éléments sont du même type et détermine si l'ensemble de leurs éléments est le même, peu importe le nombre d'occurrences.

Terminaison de la sous-fonction récursive : la taille du premier argument est un variant. La terminaison globale s'en déduit toute seule.

Complexité : Puisque la fonction `List.mem` est en temps linéaire en la taille de la liste, chaque appel récursif est linéaire en la taille du deuxième argument, pour une complexité totale de l'ordre du produit des tailles des listes (deux fois, mais le facteur constant disparaît avec la notation de Landau).

Correction : Mieux vaut prouver que la fonction `aux` détermine si son premier argument est inclus dans le deuxième plutôt que de faire une preuve globale pour la fonction `verif3`. Dans ce cas, on le fait par récurrence sur la taille du premier argument.

Septième fonction

Spécification : À première vue, on rassemble arbitrairement deux listes en une. C'est un cas particulier intéressant de fonction où la spécification va être plus précise si on fait une hypothèse sur l'entrée. En particulier, la fonction prend en argument deux listes dont les éléments sont du même type et en retourne la fusion, qui reste croissante si les deux listes en argument le sont.

Terminaison : La somme des tailles des listes en argument est le variant le plus pertinent. En plus elle décroît exactement d'un à chaque appel récursif imbriqué.

Complexité : Ainsi, grâce à ce variant, on obtient aisément une complexité linéaire en la somme des tailles des arguments.

Correction : Pour se faciliter la vie, on fera une récurrence sur la taille de la liste obtenue, avec la propriété qu'elle contient à tout moment les plus petits éléments des deux listes initiales (dans l'hypothèse où les deux listes en argument sont triées). Le plus petit élément non encore transféré est alors forcément en tête d'une des listes.