Conjecture et preuve d'un algorithme récursif

Le problème : On veut savoir ce que renvoie un programme récursif donné, puis prouver notre conjecture.

Prenons l'exemple du programme suivant :

OCaml
```
let rec f = function
        | [] -> [],[]
        | [x] -> [x],[]
        | x::y::q -> let l1,l2 = f q in x::l1,y::l2;;
```

Etape 1 : On commence par **établir une conjecture**.

→ On commence par typer la fonction :

Suite de l'exemple : `f : 'a list -> 'a list * 'a list`

→ Puis, on applique la fonction `f` à de petits arguments de taille croissante.
Le mieux est de consigner les résultats obtenus dans un tableau :

Suite de l'exemple

Valeur de l'argument l	Résultat en sortie $f(l)$
`[]`	`[],[]`
`[a]`	`[a],[]`
`[a;b]`	`[a],[b]`
`[a;b;c]`	`[a;c],[b]`

On établit alors notre conjecture :

Suite de l'exemple

Il semblerait que f décompose une liste en deux listes :

$$\begin{cases} \text{l'une contenant les éléments d'indice pair} \\ \text{l'autre contenant les éléments d'indice impair} \end{cases}$$

Etape 2 : Il faut maintenant **prouver cette conjecture**

La preuve d'un algorithme consiste à vérifier qu'il se termine et renvoie le bon résultat.
Dans le cas des fonctions récursives, on distingue deux parties dans la preuve :

1. La preuve de la terminaison
2. La preuve de la correction

- Preuve de terminaison : On veut vérifier que ce algorithme ne boucle pas infiniment

  Pour cela, on regarde comment évolue la taille des arguments dans la formule de récursivité

  Suite de l'exemple

  ⋆ Ici, on constate qu'à chaque appel récursif, la taille de l'argument diminue de 2.
  Au bout d'une nombre fini d'itérations, on aboutit nécessairement à une liste de taille 0 ou 1.

  ⋆ On constate que ces deux cas sont bien traités comme des cas de base dans la programmation de f.
  L'algorithme proposé se termine donc bien.

- Preuve de correction :

  Il s'agit de montrer que l'algorithme renvoie bien le bon résultat.
  Pour cela, on effectue un raisonnement dit « inductif » (généralisation d'une récurrence) de la forme :

  → Vérification des cas de base :

  On s'assure que la fonction renvoie le bon résultat pour chacun des cas de base.

  Suite de l'exemple : C'est bien le cas ici.

  → Hérédité :

  On suppose que les appels récursifs renvoient les bons résultats
  et on vérifie que c'est alors également le cas pour la fonction.

  Suite de l'exemple

  ⋆ Considérons la liste l = [x0 ; x1 ; x2 ; ... ; xn].

  ⋆ On suppose que `f([x2 ; ... ; xn])` renvoie bien `[x2 ; x4 ; ... ]`, `[x1 ; x3 ; ... ]`.

  ⋆ Vérifions alors ce que renvoie `f([x0 ; x1 ; ... ; xn])`.
  La formule de récursivité donne `x0::[x2 ; ... ; xn]`, `x1::[x3 ; ... ; xn]` c'est à dire :

  ```
  [x0 ; x2 ; ... ; xn], [x1 ; x3 ; ... ; xn]
  ```

  C'est bien ce que nous avions conjecturé !

« Nous avons bien prouvé notre conjecture ! »

Un exemple : Que fait le programme suivant ?

OCaml
```
let rec carthy n = match n > 100 with              (* n est un entier relatif *)
                   | true -> n-10
                   | _    -> carthy (carthy (n + 11));;
```

- typage : ?
- Conjecture : ?
- Terminaison : ?

- Correction : ?
  - → Initialisation : ?
  - → Hérédité : ?

- typage : `carthy : int -> int`

- Conjecture : On applique la fonction à des valeurs simples

  - → On constate que lorsque $n > 100$, cette fonction renvoie $n - 10$.
  - → Sinon ?

```
carthy 100 -> carthy (carthy 111) -> carthy 101 -> 91
carthy 99 -> carthy (carthy 110) -> carthy 100 -> 91
carthy 98 -> carthy (carthy 109) -> carthy 99 -> 91
```

Conjecture : Il semblerait que :
- `carthy n` renvoie `n-10` lorsque `n > 100`
- `carthy n` renvoie `91` lorsque `n <= 100`

- Terminaison : Pas facile de justifier ici de la stricte croissance des arguments.

  En effet :
  - → Il y a bien croissance des arguments dans `carthy (n+11)` car $n + 11 > n$
  - → Mais dans le deuxième appel récursif `carthy (carthy (n+11))`, il faudrait avoir une idée de la valeur de `carthy (n+11)` pour affirmer que `carthy (n+11) > n`

  Exceptionnellement dans cet exemple, la terminaison doit donc se prouver en même temps que la correction !

- Correction :

  - → Initialisation : La fonction termine et renvoie bien `n-10` lorsque $n > 100$.

  - → Hérédité : Soit $n \leq 100$.

    On suppose que `carthy m` termine et renvoie bien les bonnes valeurs pour $m > n$.

    Le premier appel récursif est `carthy (n + 11)`. Il faut donc distinguer les cas où $\begin{cases} n+11 > 100 \text{ cad } n > 89 \\ n+11 \leq 100 \text{ cad } n \leq 89 \end{cases}$.

    ⋆ Lorsque $n \in [\![90, 100]\!]$:

    `carthy n -> carthy (carthy (n + 11)) -> carthy (n+11)-10 -> carthy n+1 -> 91` par HR

    ⋆ Lorsque $n < 90$:

    `carthy n -> carthy (carthy (n + 11)) -> carthy (91) -> 91` par HR

CONCLUSION : La fonction `carthy` se termine bien pour tout $n \in \mathbb{Z}$ et :

- `carthy n` renvoie `n-10` lorsque `n > 100`
- `carthy n` renvoie `91` lorsque `n <= 100`

A vous de jouer !

Exercice 1 Conjecturer le résultat et prouver le programme suivant :

OCaml
```
let rec f = function
          | [x] -> true
          | x::y::q -> if (x > y) then false
                                  else f (y::q);;
```

Exercice 2

OCaml
```
let rec acker = function
        | (0,p) -> p + 1
        | (n,0) -> acker ((n - 1),1)
        | (n,p) -> acker (n-1,acker (n,p - 1)) ;;
```

1. Donner les expressions explicites de `acker(1,p)`, `acker(2,p)` et `acker(3,p)`.
2. Que vaut `acker(4,4)` ?
   Comparer cet entier avec 10^{80} qui est une estimation du nombre d'atomes dans l'univers.

Exercice 3

OCaml
```
let rec v = fun
     |0 -> 1
     |n -> n - v(v (n-1));;      (* n est ici un entier naturel non nul *)
```

1. Montrer que la fonction v ne se termine pas dans certains cas.
2. En revanche, prouver qu'elle termine toujours si l'on remplace l'instruction `| 0 -> 1` par : `| 0 -> 0`.