CM 3 : Algdem

Démonstrations sur les listes

```haskell
fusionne :: [Int] → [Int] → [Int]
fusionne [] l = l
fusionne l [] = l
fusionne (x1:l1) (x2:l2) | x1 ⩽ x2 = x1 : fusionne l1 (x2:l2)
                         | otherwise = x2 : fusionne (x1:l1) l2
```

> ⓘ Note
>
> Comment formaliser que deux listes contiennent les mêmes éléments ?

Ici, nous prendrons la définition suivante : le prédicat `perm`

`l1` et `l2` ont les mêmes éléments si `perm(l1, l2)`

où `perm` est une relation d'équivalence (transitivité, réflexivité, symétrique), et :

- Si `perm(l1, l2)`, alors `perm(x:l1, x:l2)`
- Si `perm(l1, l1')` et `perm(l2, l2')`, alors `perm(l1 ++ l2, l1' ++ l2')`
- `perm(x:(l1 ++ l2), l1 ++ (x:l2))`

On veut montrer :

1. (Pre1) : `l1` et ordonnée par ordre croissant (`triee(l1)`) et
2. (Pre2) : `l2` est ordonnée par ordre croissant

Alors :

1. (Post1) `triee(fusionne l1 l2)`
2. (Post2) `perm(fusionne l1 l2, l1 ++ l2)`

Le variant

On prend pour variant `length l1 + length l2`

Hypothèse d'induction

Pour toutes `l1', l2'` tel que `length l1' + length l2' < length l1 + length l2`, alors :

- `triee(fusionne l1 l2)`
- `perm(fusionne l1 l2, l1 ++ l2)`

Il y a 4 cas :

Cas 1

On a `l1 = []`, donc :

`fusionne l1 l2 = l2` (par définition)

Or, on a `triee(l2)` par hypothèse, et on a également `perm(l2, l2)`. Donc nous avons montré la spécification pour le cas 1

Cas 2

Le cas 2 est symétrique

Cas 3

On a `l1 = x1:l1'` et `l2 = x2:l2'` avec `x1 <= x2`

On a donc :

```
fusionne l1 l2 = x1 : fusionne l1'
```

> ⓘ Note
>
> On vérifie que le variant décroit, puis que `l1'` et `l2'` vérifient les pré conditions

Puisque `length l1' < length l1`, on a : `length l1' + length l2' < length l1 + length l2`

On a que `l2'` est triée par définition, et aussi `l1'`

Il faut montrer que la liste est triée :

Comme `l1` est triée, `x1` est plus petit que tous les éléments de `l1'`, et comme `x1 <= x2`, on a que `x1` est plus petit que tous les éléments de `l2` Donc, `x1` est le plus petit des éléments de `l1' ++ l2`

Comme `fusionne l1' l2` est triée, rajouter un élément plus petit que tous les élements de `l1' ++ l2` ne change pas l'ordre, donc c'est ok

Donc, `fusionne l1 = l2` est bien triée

Cas 4

Le cas 4 est symétrique au cas 3, donc on peut conclure

Conclusion

Nous avons montré pour tous les cas, donc la fonction respecte sa spécification

Récursivité terminale

Une fonction est récursive terminale lorsque dans tous les cas :

- `f` termine et renvoie une valeur
- La valeur de retour de la fonction est la valeur de retour d'un appel récursif

On peut transformer une fonction récursive terminale en itérative :

```python
def div_aux(a, b, acc):
    if a < b:
        return (acc, a)
    else:
        return div_aux(a-b, b, acc + 1)
```

En :

```python
def div_aux(a, b, acc):
    while True:
        if a < b:
            return (acc, a)
        else:
            a0 = a
            b0 = b
            acc0 = acc
            a = a0 - b0
            b = b
            acc = acc0 + 1
```

Version sans variable supplémentaire

```python
def div_aux(a, b, acc):
    while True:
        if a < b:
            return (acc, a)
        else:
            a = a - b
            acc = acc + 1
```

Cas général

On convertit généralement :

```haskell
f :: a1 → ... → a_n
f x1 ... x_n | cond1 x1 = e1
             | ...
             | condn x1 ... n_n = e_n
```

en :

```python
def f(x1, ..., x_n):
    while True:
        if cond1(x1, ..., x_n):
            return e1
        ...
        if condn(x1, ..., x_n):
            return e_n
```

Une fonction non récursive terminale renvoie un résultat de la forme : `h(f x1 ... x_n)`

Exemple

```haskell
divAux :: Int → Int → Int → (Int, Int)
divAux a b acc | a < b = (acc, a)
               | otherwise = divAux (a-b) b (acc + 1)
```

On a que `divAux n b 0 = divAux a b acc = (q, r)`

On a que `n = q * b + r` et `0 <= r < b`

`a = n - acc * b => n = a + acc * b`

Donc :

`a + acc*b  = q*b + r` et `0 <= r < b`

Démonstration

- Pré condition : `a >= 0, b > 0`
- Post condition : en posant `(q, r) = divAux a b acc`, on a:
  - `acc * b + a = q*b+r`
  - `0 <= r < b`
- Variant : `a`

Hypothèse d'induction

Pour tout `a', b', acc'` tel que `a' < a`, `a' >= 0` et `b' > 0`, en posant `(q', r') = divAux a' b' acc'`, on a que `acc'*b' + a' = q' * b' + r'` avec `0 <= r' < b'`

Cas 1

`a < b`

Dans ce cas, on a `divAux a b acc = divAux (a - b) b (acc + 1)`. On montre que `acc * b + a = acc * b + a` facile. Il faut que `0 <= a < b`. Or , on a `a >= 0` et on a également `a < b`

Cas 2

`a >= b`

On a `divAux a b acc = divAux (a - b) b (acc + 1)`.

On a : `a-b < a` puisque `b > 0`, donc le variant décroit

Il faut maintenant montrer que `a - b >= 0`, ce qui est vrai puisque `a >= b`, on a aussi `b > 0`, nous pouvons donc appliquer l'hypothèse d'induction

on a alors :
`divAux (a - b) b (acc + 1) = (q', r') => acc'*b' + a' = q' * b' + r et 0 <= r' < b'` (j'ai perdu le fil)