Correction

« Beware of bugs in the above code ; I have only proved it correct, not tried it. »

— Donald Knuth (1938–)

THE PROBLEM WITH AVERAGING STAR RATINGS

Table des matières

1 Correction

1.1 Spécification d'une fonction

La *spécification* d'une fonction, c'est le contrat qu'elle doit respecter. On peut la découper ainsi :

— *Entrées :*
  — *Nombre et types des arguments* : par exemple, « cette fonction prend en entrée une liste t d'entiers et un entier x ».
  — *Préconditions* : une ou plusieurs conditions qui doivent être vérifiées par les entrées pour que la fonction s'exécute correctement. Par exemple, « les éléments de la liste doivent être distincts », « la liste doit être triée par ordre croissant », « l'entier passé en argument est positif », etc. Si ces préconditions ne sont pas vérifiées, la fonction peut avoir un comportement arbitraire. Autrement dit, elle fait ce qu'elle veut, par exemple renvoyer un résultat arbitraire, planter, formater le disque dur, envoyer la totalité des images présentes sur votre ordinateur à tous vos contacts de messagerie, etc.
— *Sorties :*

— *Type du résultat* : par exemple, « cette fonction renvoie un entier », ou « cette fonction renvoie un tuple formé d'un entier et d'une liste d'entiers », ou « cette fonction renvoie `None` ».
— *Valeur du résultat* : si la fonction renvoie une « vraie » valeur (différente de `None`), que doit vérifier cette valeur ? Par exemple, « la fonction renvoie $n!$ où n est l'argument » ou « la fonction renvoie une liste contenant les mêmes éléments que l'argument, mais triée par ordre croissant ».
— *Effets secondaires* (éventuellement) : Tous les effets que la fonction a sur le monde, en dehors du renvoi de son résultat. Typiquement, modification de l'argument ou d'une variable globale, affichage sur l'écran, suppression de toutes les données de l'ordinateur, envoi de missiles intercontinentaux, etc. Par exemple : « après l'exécution de la fonction, le tableau passé en argument est trié et contient les mêmes éléments qu'au départ ».

Remarques

⇨ On peut regrouper « effets secondaires » et « valeur du résultat » sous le terme *postconditions*.

⇨ Une fonction ne doit pas avoir d'effets secondaires autres que ceux apparaissant dans sa spécification. Par exemple, la fonction suivante n'est pas une manière acceptable de calculer le maximum d'un tableau.

```
def apres_moi_le_deluge(t):
    """apres_moi_le_deluge(t: list[int]) -> int"""
    for i in range(1, len(t)):
        t[i] = max(t[i], t[i - 1])
    return t[-1]
```

En effet, son exécution ne laisse pas le système dans un état acceptable :

```
In [1]: [7, 2, 9, 0]

In [2]: apres_moi_le_deluge(t)
Out[2]: 9

In [3]: t
Out[3]: [7, 7, 9, 9]
```

⇨ En pratique, on essaie souvent d'éviter les comportements totalement arbitraires. S'il y a un moyen simple et efficace de vérifier que les préconditions sont remplies, on peut préférer faire ces tests pour détecter un éventuel problème le plus tôt possible. Pour cela, on utilisera le mot clé `assert` déjà utilisé pour les tests unitaires. Par exemple, pour une fonction calculant la factorielle, on écrit :

```
def factorielle(n):
    """factorielle(n: int) -> int"""
    assert n >= 0
    f = 1
    for i in range(1, n + 1):
        f = f * i
    return f
```

Ainsi, un appel sur un entier $n < 0$ sera immédiatement détecté comme une erreur :

```
In [1]: factorielle(-1)
AssertionError:
```

Sans le `assert`, la fonction aurait renvoyé un résultat dénué de sens, 1 en l'occurrence, sans se plaindre, ce qui aurait rendu l'erreur beaucoup plus difficile à détecter.

⇨ Attention cependant, ce conseil n'est pas toujours possible à appliquer, car la précondition peut être trop couteuse, voire impossible, à vérifier. Par exemple, si l'on effectue une recherche dichotomique dans une liste triée, algorithme qui s'exécute dans le pire des cas en $\Theta(\log n)$, il serait absurde de vérifier que la liste est bien triée car cette opération a une complexité en $\Theta(n)$. Enfin, il faut avoir conscience que ces assertions ne sont utiles qu'en « production » et il est bien entendu inutile d'en placer dans un écrit de concours, sauf si cela vous est explicitement demandé.

1.2 Correction partielle, correction totale

Définition 1.1: Terminaison d'une fonction

On dit qu'une fonction *termine* lorsqu'elle renvoie un résultat en un nombre fini d'étapes, quelles que soient les valeurs de ses paramètres vérifiant les préconditions.

Remarque

⇨ Le nombre d'étapes de calcul ne sera en général pas *borné*, puisqu'il dépend de la valeur des paramètres.

Exemple

⇨ La fonction suivante calcule $n!$ pour $n \geqslant 0$.

```
def factorielle(n):
    """factorielle(n: int) -> int"""
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)
```

On considère qu'elle termine, car il y a une précondition $n \geqslant 0$. Cependant, si on l'appelle sur un entier $n < 0$, l'appel récursif ne termine pas.

Définition 1.2: Correction partielle

Une fonction est dite *partiellement correcte* par rapport à sa spécification lorsque, quelles que soient les valeurs des paramètres vérifiant les préconditions :
- Soit elle renvoie un résultat conforme à la spécification.
- Soit elle ne termine pas.

Définition 1.3: Correction totale

Une fonction est dite *totalement correcte*, ou simplement *correcte*, si elle est partiellement correcte et qu'elle termine.

Exemples

⇨ La fonction suivante est partiellement correcte, vis-à-vis de n'importe quelle spécification : il n'y a aucun risque qu'elle renvoie un résultat incorrect.

```
def f(x):
    return f(x)
```

⇨ La fonction suivante est censée calculer x^n pour $x \in \mathbb{Z}$ et $n \in \mathbb{N}$.

```
def puissance(x, n):
    """puissance(x: int, n: int)"""
    if n == 1:
        return x
    else:
        return x * puissance(x, n - 1)
```

Elle est partiellement correcte, mais pas totalement correcte : en effet, `puissance(x, 0)` ne termine pas.

2 Algorithme itératif

2.1 Terminaison

La preuve de la terminaison d'un programme n'utilisant que des boucles `for` est immédiate. En présence de boucles `while`, prouver la terminaison peut être arbitrairement compliqué : ce problème est *indécidable*, c'est-à-dire qu'il n'existe pas d'algorithme permettant de déterminer si un programme quelconque termine. Cela n'empêche pas de montrer la terminaison dans de nombreux cas particuliers.

Définition 2.1: Variant de boucle

Un *variant de boucle* est une quantité :
- entière
- minorée
- qui décroit *strictement* à chaque passage dans une boucle.

Proposition 2.2

Si une boucle admet un variant de boucle, alors elle termine.

Remarque

⇨ Une quantité entière, *majorée* et qui *croît* strictement fait aussi l'affaire.

Exemple

⇨ Considérons la fonction suivante.

```
def log2(n):
    """log2(n: int) -> int"""
    i = 0
    x = n
    while x > 1:
        x = x // 2
        i = i + 1
    return i
```

Alors x est un variant de boucle.
- C'est un entier.
- Il est minoré par 1, tant qu'on est dans la boucle.
- En notant x' la valeur en fin d'itération, on a $x' \coloneqq \lfloor x/2 \rfloor \leqslant x/2 < x$ puisque $x \geqslant 1$, donc il est strictement décroissant.

Cette fonction termine donc. Attention, si l'on remplace la condition de la boucle par `x >= 0`, la terminaison n'est plus assurée, car la décroissance n'est plus stricte.

Exercice 1

⇨ On considère la fonction suivante.

```
def disjoints(u, v):
    """disjoints(u: list[int], v: list[int]) -> bool"""
    iu = 0
    iv = 0
    nu = len(u)
    nv = len(v)
    while iu < nu and iv < nv and u[iu] != v[iv]:
        if u[iu] < v[iv]:
            iu = iu + 1
        else:
            iv = iv + 1
    return iu == nu or iv == nv
```

1. `iu` est-il un variant de boucle ? Même question pour `iv`.
2. Identifier un variant de boucle.
3. Quelle précondition doit être vérifiée par `u` et `v` pour que cette fonction soit « correcte ». Il faut bien sûr commencer par préciser ce que *correcte* signifie ici, en s'aidant du nom de la fonction.

Les variants de boucle ne sont pas le seul outil : fondamentalement, tout type de raisonnement peut être utilisé pour prouver la terminaison d'une fonction.

Exemples

⇨ Considérons la fonction ci-dessous.

```
def inv_fact(n):
    """inv_fact(n: int) -> int"""
    i = 0
```

```
    f = 1
    while f < n:
        i = i + 1
        f = f * i
    return i
```

Une récurrence simple montre qu'après k passages dans la boucle, i vaut k et f vaut $k!$. Comme $k! \to +\infty$, il est alors « clair » que ce programme termine pour toute valeur de n.

⇨ Considérons maintenant le programme suivant.

```
def syracuse(n):
    """syracuse(n: int) -> int"""
    k = n
    i = 0
    while k != 1:
        i = i + 1
        if k % 2 == 0:
            k = k // 2
        else:
            k = 3 * k + 1
    return i
```

Si vous arrivez à montrer qu'il termine pour toute valeur de n, faites-moi signe.

2.2 Correction

Les preuves de correction simples de programmes itératifs reposent sur le principe d'*invariant de boucle*, idée très similaire à celle d'une récurrence mathématique.

Définition 2.3: Invariant de boucle

Un *invariant de boucle* est un *prédicat* $\mathcal{I}$ ayant les propriétés suivantes :
- il est vrai avant de rentrer dans la boucle
- si il est vrai au début d'une itération, il reste vrai à la fin de cette itération.

Remarques

⇨ Un invariant de boucle n'a cependant aucune raison d'être vrai en milieu d'itération.

⇨ Pour une boucle *conditionnelle* (boucle `while`) avec une condition $\mathcal{C}$ et un invariant $\mathcal{I}$, on commence par prouver que $\mathcal{I}$ est vérifié avant de rentrer dans la boucle, puis on montre l'hérédité, c'est-à-dire que $(\mathcal{C}$ et $\mathcal{I}) \longrightarrow \mathcal{I}$: si la condition de boucle et l'invariant sont vérifiés en début d'itération, alors l'invariant est vérifié en fin d'itération. En sortie de boucle, (non $\mathcal{C}$) et $\mathcal{I}$ seront alors vrais.

⇨ Pour une boucle *inconditionnelle* (boucle `for`), on rédigera la preuve d'invariant en incorporant l'indice de boucle au prédicat. Pour effectuer la correction d'une boucle du type « `for k in range(a, b)` », on définit les prédicats $\mathcal{I}_a, \dots, \mathcal{I}_b$ et on prouve que :
- $\mathcal{I}_a$ est vérifié avant de rentrer dans la boucle.
- Pour tout $k \in [\![a, b[\![$, si $\mathcal{I}_k$ est vérifié au début de l'itération d'indice k, $\mathcal{I}_{k+1}$ est vérifié en fin d'itération.

En sortie de boucle, $\mathcal{I}_b$ sera alors vrai.

⇨ Comme pour la terminaison, les preuves de correction peuvent être arbitrairement difficiles : la correction d'un programme peut dépendre d'une conjecture mathématique, par exemple.

⇨ Dans les preuves, on notera souvent x la valeur de la variable `x` en début d'itération et x' sa valeur en fin d'itération.

Exemple

⇨ On souhaite montrer que le programme suivant renvoie bien un indice du minimum de t. Pour cela, on considère l'invariant de boucle $\mathcal{H}_i$: « $m = \min(t_0, \dots, t_{i-1})$ et `t[ind] = m` ». On note n la longueur de t.

```
def ind_min(t):
    """ind_min(t: list[int]) -> int"""
    ind = 0
    m = t[0]
    for i in range(1, len(t)):
        if t[i] < m:
            m = t[i]
            ind = i
    return ind
```

— $\mathcal{H}_1$ est vérifié avant de rentrer dans la boucle car $ind = 0$ et $m = t_0 = \min(t_0)$.
— Pour $1 \leqslant i \leqslant n-1$, en supposant $\mathcal{H}_i$ vraie en début d'itération, on a deux cas :
  — Si $t_i < \min(t_0, \ldots, t_{i-1}) = m$, alors en fin d'itération $m' = t_i$ et $ind' = i$, ce qui est correct puisqu'alors $\min(t_0, \ldots, t_i) = t_i$.
  — Sinon, on a $\min(t_0, \ldots, t_i) = \min(t_0, \ldots, t_{i-1}) = m = t_{ind}$. Or on ne fait rien dans ce cas et l'on a donc $m' = m$ et $ind' = ind$, ce qui est bien correct.

À la fin de l'exécution, $\mathcal{H}_n$ est donc vraie, c'est-à-dire $m = \min(t_0, \ldots, t_{n-1}) = \min t$ et `t[ind] = m`. La variable `ind` contient donc bien un indice du minimum de t.

En pratique, dans un cas aussi simple, on se contentera au mieux de donner l'invariant de boucle sans démonstration. Dans des cas plus compliqués, en revanche, l'invariant de boucle est indispensable.

2.3 Exemples fondamentaux

Les deux algorithmes présentés ici sont à connaitre absolument.

2.3.1 Exponentiation rapide, version itérative

On considère la fonction `expo(a: int, n: int) -> int`, dont la spécification est :

Précondition : $n \geqslant 0$

Résultat : `expo(a, n)` $= a^n$

```
def expo(a, n):
    """expo(a: int, n: int) -> int"""
    p = n
    x = 1
    b = a
    while p != 0:
        if p % 2 == 1:
            x = x * b
        b = b * b
        p = p // 2
    return x
```

1. Montrer que $x \cdot b^p = a^n$ est un invariant de boucle.
2. En déduire la correction partielle de la fonction.
3. Montrer la correction totale de la fonction.

2.3.2 Recherche dichotomique

On considère l'algorithme suivant :

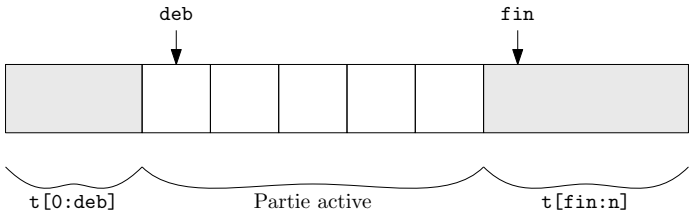
Entrées : un tableau $t = (t_0, \ldots, t_{n-1})$ et une valeur x

Précondition : t est trié par ordre croissant

Résultat : un indice $i \in [\![0, n[\![$ tel que $t_i = x$ s'il en existe un, n sinon.

Algorithme 1 Recherche dichotomique dans un tableau trié

fonction RECHERCHE(x, t)
  $deb \leftarrow 0$
  $fin \leftarrow n$
  tant que $fin - deb > 0$ **faire**
    $milieu \leftarrow (deb + fin)//2$ ▷ Division entière
    si $t_{milieu} = x$ **alors**
      renvoyer $milieu$
    sinon si $t_{milieu} < x$ **alors**
      $deb \leftarrow milieu + 1$
    sinon
      $fin \leftarrow milieu$
    fin si
  fin tant que
  renvoyer n
fin fonction



1. Montrer que cet algorithme termine.
2. Montrer que si l'algorithme renvoie un indice $i \neq n$, alors ce résultat est correct.
3. Montrer qu'on a l'invariant suivant : « $x \notin t[0 : deb] \ \cup \ t[fin : n]$ ».
4. En déduire la correction de l'algorithme.
5. L'algorithme reste-t-il totalement ou partiellement correct si
   (a) on remplace la ligne $deb \leftarrow milieu + 1$ par $deb \leftarrow milieu$?
   (b) on remplace la ligne $fin \leftarrow milieu$ par $fin \leftarrow milieu - 1$?

3 Algorithme récursif

Une première remarque s'impose : il n'y a pas de différence fondamentale entre un programme écrit à l'aide de boucles ou de manière récursive. En effet, il est facile de remplacer toutes les boucles `while` par des récursions, et toujours possible, mais pas facile en règle générale, de remplacer la récursion par une boucle `while`. En un certain sens, tout ce qui a été dit sur les programmes itératifs s'applique donc aux programmes récursifs, en particulier le caractère arbitrairement difficile des preuves de terminaison.

3.1 Principe général

En première approche, la terminaison d'un programme récursif repose sur l'existence d'un certain nombre, potentiellement infini, de cas de base et sur la certitude que toute suite d'appels finit par arriver sur l'un de ces cas de base.

Le cas le plus simple et le plus fréquent est celui d'un programme ayant une définition de ce type :
— $f(0)$ est donné.
— $\forall n \in \mathbb{N},\ f(n+1) \coloneqq g(n, f(n))$ où g est une fonction que l'on sait calculer.
Il est alors immédiat de prouver la terminaison par récurrence, et souvent possible de prouver simultanément la correction.

Exercice 2

⇨ Montrer que le programme suivant termine et calcule $n!$, pour tout entier $n \geqslant 0$.

```
def factorielle(n):
    """factorielle(n: int) -> int"""
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)
```

Fondamentalement, ce qui rend possible la récurrence est la décroissance stricte de l'argument au cours des appels récursifs. Comme $\mathbb{N}$ n'admet pas de suite infinie strictement décroissante, on peut alors conclure que la suite des appels se termine. En pratique, on justifiera cette décroissance et l'on ne rédigera pas la récurrence. C'est l'équivalent d'un variant de boucle dans un programme impératif.

La correction d'un programme récursif est parfois nettement plus simple à prouver que celle d'un programme itératif équivalent. En effet, la correspondance entre le code et les identités mathématiques qui le justifient peut apparaitre beaucoup plus clairement dans le cas récursif.

```
def expo(x, n):
    """expo(x: int, n: int) -> int"""
    if n == 0:
        return 1
    else:
        if n % 2 == 0:
            return expo(x * x, n // 2)
```

```
        else:
            return x * expo(x, n - 1)
```

— Si $n > 0$, $0 \leqslant n-1 < n$ et $0 \leqslant n/2 < n$, donc n décroit strictement au cours des appels jusqu'à être nul : la fonction termine.
— Les identités $x^0 = 1$, $x^{2k} = (x^2)^k$ et $x^n = x \cdot x^{n-1}$ si $n \geqslant 1$ sont correctes, donc la fonction aussi.

Il arrive souvent que la suite des arguments ne soit pas strictement décroissante, soit parce que cela n'a pas de sens, soit parce que c'est simplement faux. Dans ce cas, on peut chercher une fonction φ de l'ensemble des arguments dans $\mathbb{N}$ dont les valeurs décroissent strictement au cours des appels successifs. Autrement dit, on cherche une fonction φ à valeurs dans $\mathbb{N}$ telle que, dès que le programme f contient un appel du type $f(x) \to f(y)$, on ait $\varphi(y) < \varphi(x)$.

Exercice 3

⇨ Justifier la terminaison de la fonction suivante.

```
def f(n):
    """f(n: int) -> int"""
    if n % 2 != 0 or n == 0:
        return n
    else:
        return f((3 * n) // 2)
```