1 Spécification d'un programme

Vous venez de terminer l'écriture du programme suivant :

```
>>> def pgcd(a, b):
        while b > 0:
            a, b = b, a % b
        return a
```

Il a de nombreuses qualités. Tout d'abord, il porte un nom compréhensible pour toute personne s'intéressant à l'arithmétique. Ensuite, il a le bon gout de renvoyer un résultat en accord avec son nom lorsque vous l'essayez sur des exemples simples. Par exemple `pgcd(15, 21)` renvoie bien 3. Satisfait de votre travail, vous décidez d'en tirer un revenu substantiel en commercialisant votre dur labeur. Après quelques années prospères à la tête de la société EUCLID & SONS, vous recevez un courrier du dénommé C. Burns, directeur de centrale nucléaire. Il vous assigne en justice, car votre fonction est responsable d'un incident dans son installation. Lors de la dernière rotation des barres d'uranium, le programme de la centrale a dû calculer `pgcd(15, -21)` et votre programme a renvoyé 15. Depuis, nous avons perdu tout contact avec la région de Springfield. Un procès de grande ampleur vous attend.

Pour éviter une telle déconvenue, il est nécessaire de spécifier votre programme. Spécifier, c'est établir un contrat entre la personne qui publie un programme et celle qui l'utilise, le client. Le client s'engage à fournir des données au programme qui vérifient les *préconditions*. En échange, le programme s'engage à produire des valeurs qui vérifient les conditions de sorties, appelées *postconditions*. Par exemple, la précondition de la fonction `pgcd` est que les valeurs a et b passés en entrée soient des entiers positifs. La postcondition est que la valeur renvoyée est le pgcd de a et b. Pour spécifier ce programme, on écrira donc :

```
>>> def pgcd(a, b):
        # precondition {a:int & b:int & a >= 0 & b >= 0}
        # postcondition {resultat:int & resultat = pgcd(a, b)}
        while b > 0:
            a, b = b, a % b
        return a
```

Les « : » suivi d'un type permettent de donner l'hypothèse que l'on fait sur le type des valeurs d'entrée et de préciser le type de la valeur renvoyée. Comme la spécification de programmes mettant en jeu des nombres flottants est très délicate, on se limitera le plus souvent aux entiers et on omettra de spécifier les types.

Une fois qu'un programme est spécifié, il est nécessaire de valider le programme, c'est-à-dire s'assurer qu'il répond à cette spécification. On devra s'assurer de sa *correction* et de sa *terminaison*. Valider la correction d'un algorithme vis-à-vis de sa spécification, c'est montrer que si la précondition est remplie et que l'algorithme termine, alors la postcondition est vérifiée après son exécution. Valider sa terminaison, c'est montrer que si la précondition est remplie, l'algorithme termine. Pour valider des fonctions, la technique la plus courante est d'écrire des *tests unitaires*. Ce sont des programmes qui testent notre fonction et qui vérifient qu'elle donne la bonne réponse sur une liste de cas tests judicieusement choisis. Le programme suivant

```
>>> def test_unitaire_pgcd(f):
        test0 = f(0, 0) == 0
        test1 = f(0, 1) == 1
        test2 = f(1, 0) == 1
        test3 = f(2, 3) == 1
        test4 = f(15, 21) == 3
        return test0 and test1 and test2 and test3 and test4
```

vérifie quelques cas simples. Il suffit d'appeler `test_unitaire_pgcd(pgcd)` et de vérifier que le test renvoie bien `True`. Cependant, la fonction `pgcd` peut très bien contenir un bug qui n'est pas mis en valeur par nos tests. Les tests unitaires ne peuvent donc en aucun cas certifier la correction de notre fonction. Ils peuvent seulement détecter certains bugs. Néanmoins, ils ont un très grand avantage : il est très facile de les lancer pour vérifier qu'une mise à jour n'a pas introduit d'erreur manifeste.

Une autre méthode pour valider un programme est de le *prouver* : par un raisonnement logique, on prouve que le programme termine et donne la bonne réponse. On parle de preuve de *terminaison* et de *correction* du programme. Cette méthode demande beaucoup d'efforts et de temps pour des programmes conséquents. C'est la raison pour laquelle on l'emploie essentiellement pour prouver de nouveaux algorithmes ou des applications critiques. Cependant, les programmes que nous écrirons cette année sont suffisamment simples pour que leur spécification et leur preuve soient à notre portée.

2 Spécification d'un algorithme

Un algorithme est une suite finie et non ambigüe d'opérations ou d'instructions permettant de résoudre une classe de problèmes. Afin de spécifier un algorithme, on doit définir :

- *Entrées* : Les entrées sont des valeurs qui sont données avant qu'un algorithme commence. Ces valeurs vérifient des propriétés qu'on appelle *préconditions*.
- *Sorties* : Les sorties sont des valeurs produites par l'algorithme. Elles vérifient des propriétés en lien avec les entrées qu'on appelle *postconditions*.

Analyser un algorithme, c'est prouver sa *terminaison* et sa *correction*. Prouver sa *terminaison* consiste à démontrer que, quelles que soient les entrées vérifiant les préconditions, l'algorithme termine en un nombre fini d'opérations. Prouver sa *correction* consiste à prouver que les données produites en sorties vérifient les postconditions.

Prenons l'exemple de l'algorithme suivant :

```
>>> def factorielle(n):
        ans = 1
        for k in range(1, n + 1):
            ans = ans * k
        return ans
```

Cet algorithme possède une entrée : n. La précondition est que n est un entier naturel. La postcondition est que la valeur renvoyée par cette fonction est l'entier $n!$. On prouvera dans cette partie la *terminaison* et la *correction* de cet algorithme.

La correction et la terminaison d'un programme impératif est délicate. Pour de tels programmes, la précondition et la postcondition sont des prédicats sur l'état du système. Par exemple, lorsqu'on écrit

```
>>> # precondition: {n = n0 & n >= 0}
    p = 1
    for k in range(1, n + 1):
        p = p * k
    # postcondition: {p = n0!}
```

on dit que quel que soit l'état du système pour lequel la variable `n` contient un entier positif, après l'exécution du code, le système se trouve dans un état dans lequel `p` contient sa factorielle. L'utilisation de n_0 est nécessaire pour capturer la valeur de `n` avant l'exécution du code et l'utiliser dans la postcondition. On comprend mieux son intérêt lorsqu'on change la valeur des variables dans le code :

```
>>> # precondition {a = a0 & b = b0 & a >= 0 & b >= 0}
    while b != 0:
        a, b = b, a % b
    # postcondition {a = pgcd(a0, b0)}
```

Pour prouver qu'un code est correct vis-à-vis des préconditions et des postconditions données par les spécifications, nous allons utiliser des invariants.

3 Boucle conditionelle

On cherche à prouver des préconditions et des postconditions pour le code suivant :

```
>>> # precondition P
    while <<Condition de boucle>>:
        <<Corps de la boucle>>
    # postcondition Q
```

Nous savons que la boucle va s'exécuter tant que la condition de boucle $\mathcal{C}$ est vraie. Cette condition définit d'ailleurs un prédicat sur l'état du système.

On appelle invariant de boucle tout prédicat $\mathcal{I}$ sur l'état du système vérifiant les 3 points suivants :

— $\mathcal{P} \Longrightarrow \mathcal{I}$: Si un état du système vérifie $\mathcal{P}$, alors il vérifie $\mathcal{I}$.

— ($\mathcal{I}$ et $\mathcal{C}$) $\longrightarrow \mathcal{I}$: Si un état du système vérifie $\mathcal{I}$ et $\mathcal{C}$, alors l'exécution du corps de la boucle va le transformer en un état vérifiant $\mathcal{I}$.

— ($\mathcal{I}$ et (non $\mathcal{C}$)) $\Longrightarrow \mathcal{Q}$: Si un état du système vérifie $\mathcal{I}$ mais pas $\mathcal{C}$, alors il vérifie $\mathcal{Q}$.

La donnée d'un invariant de boucle permet de prouver la *correction partielle* de la boucle vis-à-vis de ses spécifications, c'est-à-dire que si on sort de la boucle, alors la postcondition est vérifiée. En effet, on note $\mathcal{E}_0$ l'état du système avant d'entrer dans la boucle. Puisqu'il vérifie la précondition $\mathcal{P}$, il vérifie $\mathcal{I}$. Si la condition $\mathcal{C}$ n'est pas vérifiée, on sort de la boucle. L'état en sortie de boucle vérifie ($\mathcal{I}$ et (non $\mathcal{C}$)) et donc $\mathcal{Q}$. Si la condition $\mathcal{C}$ est vérifiée, on exécute le corps de la boucle. On se trouve dans la condition de l'invariant et on en déduit que l'état $\mathcal{E}_1$ du système après la boucle vérifie $\mathcal{I}$. On regarde alors si la condition $\mathcal{C}$ est vérifiée, etc.

$$\mathcal{E}_0 \longrightarrow \mathcal{E}_1 \longrightarrow \cdots \longrightarrow \mathcal{E}_k \longrightarrow \mathcal{E}_{k+1} \longrightarrow \cdots \longrightarrow \mathcal{E}_n$$

De manière générale, si la condition $\mathcal{C}$ sur l'état $\mathcal{E}_k$ n'est pas vérifiée, on sort de la boucle. L'état à la sortie vérifie $\mathcal{I}$ mais pas $\mathcal{C}$, donc $\mathcal{Q}$. Mais si la condition est vérifiée, on est dans les conditions de l'invariant et donc l'exécution du corps de la boucle va produire un état $\mathcal{E}_{k+1}$ qui vérifie $\mathcal{I}$. Si la boucle s'arrête après n itérations, on sort dans un état qui vérifie ($\mathcal{I}$ et (non $\mathcal{C}$)), donc $\mathcal{Q}$. Si elle ne s'arrête pas, on est parti en boucle infinie.

Prenons le cas concret suivant :

```
>>> # precondition {a = a0 & b = b0 & a >= 0 & b >= 0}
    while b != 0:
        a, b = b, a % b
    # postcondition {a = pgcd(a0, b0)}
```

pour lequel on va montrer que

$$\mathcal{I} : \text{« } \mathrm{pgcd}(a,b) = \mathrm{pgcd}(a_0,b_0) \text{ et } a \geqslant 0 \text{ et } b \geqslant 0 \text{ »}$$

est un invariant de boucle.

— $\mathcal{P} \Longrightarrow \mathcal{I}$: Supposons que $a = a_0$, $b = b_0$, $a \geqslant 0$ et $b \geqslant 0$. Alors $\mathrm{pgcd}(a,b) = \mathrm{pgcd}(a_0,b_0)$, donc $\mathcal{I}$ est bien vérifié.

— ($\mathcal{I}$ et $\mathcal{C}$) $\longrightarrow \mathcal{I}$: Supposons que $\mathrm{pgcd}(a_k,b_k) = \mathrm{pgcd}(a_0,b_0)$, $a_k \geqslant 0$, $b_k \geqslant 0$ et $b_k \neq 0$. On effectue la division euclidienne de a_k par b_k donc il existe $q \in \mathbb{Z}$ et $b_{k+1} \in \mathbb{N}$ tels que $a_k = qb_k + b_{k+1}$ et $0 \leqslant b_{k+1} < b_k$. Or $a_{k+1} = b_k$ donc $\mathrm{pgcd}(a_{k+1},b_{k+1}) = \mathrm{pgcd}(b_k,b_{k+1}) = \mathrm{pgcd}(a_k,b_k)$ d'après le lemme d'Euclide. De plus a_{k+1} et b_{k+1} sont positifs. Donc $\mathcal{I}$ est vérifié en fin d'itération.

— ($\mathcal{I}$ et (non $\mathcal{C}$)) $\Longrightarrow \mathcal{Q}$: Supposons que $\mathrm{pgcd}(a_n,b_n) = \mathrm{pgcd}(a_0,b_0)$, $a_n \geqslant 0$, $b_n \geqslant 0$ et $b_n = 0$. Alors $\mathrm{pgcd}(a_0,b_0) = \mathrm{pgcd}(a_n,0) = a_n$ car $a_n \geqslant 0$. La postcondition est donc bien vérifiée.

Sur cet exemple, on voit bien d'où vient le mot invariant, puisque la quantité $\mathrm{pgcd}(a,b)$ ne change pas d'une itération à l'autre.

Il reste maintenant à prouver la terminaison de l'algorithme. Pour cela, on utilise ce qu'on appelle un *variant*, c'est-à-dire une quantité entière positive, calculée à partir de l'état du système et qui décroit strictement d'une itération à l'autre. Ici le variant est b. En effet $0 \leqslant b_{k+1} < b_k$. Ceci prouve la terminaison de la boucle puisqu'il n'existe pas de suite infinie

d'entiers positifs strictement décroissante.

On a donc prouvé la *correction totale* de l'algorithme, c'est-à-dire qu'il termine et qu'il est correct. On pourra donc écrire en commentaire :

```
>>> # precondition: {a = a0 & b = b0 & a >= 0 & b >= 0}
    while b != 0:
        # invariant {pgcd(a, b) = pgcd(a0, b0) & a >= 0 & b >= 0}
        # variant b
        a, b = b, a % b
    # postcondition {a = pgcd(a0, b0)}
```

4 Boucle non conditionnelle

On souhaite maintenant prouver des préconditions et des postconditions pour le code suivant :

```
>>> # precondition P
    for k in range(a, b):
        <<Corps de la boucle>>
    # postcondition Q
```

On appelle invariant de boucle toute famille de prédicats $\mathcal{I}_a, \dots, \mathcal{I}_b$ sur l´état du système vérifiant les 3 points suivants :

- $\mathcal{P} \Longrightarrow \mathcal{I}_a$: Si un état du sytème vérifie la propriété $\mathcal{P}$, alors il vérifie la propriété $\mathcal{I}_a$.
- $\mathcal{I}_k \longrightarrow \mathcal{I}_{k+1}$: Pour tout $k \in [\![a, b-1]\!]$, si le système est dans un état vérifiant $\mathcal{I}_k$, l'exécution du corps de la boucle dans lequel la variable `k` contient la valeur k va le transformer en un état vérifiant $\mathcal{I}_{k+1}$.
- $\mathcal{I}_b \Longrightarrow \mathcal{Q}$: Si un état du système vérifie la propriété $\mathcal{I}_b$, alors il vérifie la propriété $\mathcal{Q}$.

La donnée d'un invariant de boucle permet de prouver la correction totale de l'algorithme. La démonstration se fait demanière similaire à la preuve pour les invariants de boucle conditionnelle. L'exécution de la boucle va transformer l'état initial du système $\mathcal{E}_a$ en des états successifs $\mathcal{E}_a, \dots, \mathcal{E}_b$. Chaque état $\mathcal{E}_k$ va vérifier la propriété $\mathcal{I}_k$. L'état final vérifiera donc la propriété $\mathcal{I}_b$ et donc $\mathcal{Q}$.

Prenons l'exemple du calcul de la factorielle :

```
>>> p = 1
    # precondition {n = n0 & n >= 0 & p = 1}
    for k in range(1, n + 1):
        p = p * k
    # postcondition {p = n0!}
```

Nous allons montrer que la famille

$$\mathcal{I}_k : \text{« } p = (k-1)! \text{ »}$$

pour $k \in [\![1, n]\!]$, forme un invariant de boucle.

- $\mathcal{P} \Longrightarrow \mathcal{I}_1$: En effet si l'état initial vérifie la précondition, alors $p_1 = 1 = 0!$.
- $\mathcal{I}_k \longrightarrow \mathcal{I}_{k+1}$: Soit $k \in [\![1, n]\!]$. On suppose que le système est dans un état pour lequel $p_k = (k-1)!$. Alors l'exécution du corps de la boucle va laisser le système dans un état dans lequel $p_{k+1} = k(k-1)! = k!$.
- $\mathcal{I}_{n+1} \Longrightarrow \mathcal{Q}$: C'est immédiat.

On a donc prouvé la correction totale de l'algorithme de calcul de la factorielle.

Prouvons désormais la validité de ce programme qui calcule le minimum d'une liste non vide d'entiers.

```
>>> def minimum(t)
        # precondition {t: liste d'entiers non vide}
        # postcondition {resultat = plus petit élément de la liste}
        n = len(t)
        ans = t[0]
        for k in range(1, n):
            if t[k] < ans:
                ans = t[k]
        return ans
```

Nous allons montrer que la famille

$$\mathcal{I}_k : \text{« ans contient le minimum de la liste } t[0], \dots, t[k-1] \text{ »}$$

pour $k \in [\![1, n]\!]$, forme un invariant de boucle.

- $\mathcal{P} \Longrightarrow \mathcal{I}_1$: En effet, avant d'entrer dans la boucle, ans est égal à $t[0]$ qui est bien le minimum de la liste qui ne contient que $t[0]$.
- $\mathcal{I}_k \longrightarrow \mathcal{I}_{k+1}$: Soit $k \in [\![1, n-1]\!]$. On suppose que le système est dans un état pour lequel ans est le minimum de $t[0], \dots, t[k-1]$. Alors l'exécution du corps de la boucle va laisser le système dans un état dans lequel ans est le minimum de $t[0], \dots, t[k]$
- $\mathcal{I}_n \Longrightarrow \mathcal{Q}$: En fin de boucle ans est bien le minimum de $t[0], \dots, t[n-1]$, c'est-à-dire de toute la liste.

On a donc prouvé la correction totale de l'algorithme de recherche du minimum.

La preuve de programmes ayant plusieurs points de sortie reste possible, mais nécessite un peu plus d'attention. Prenons l'exemple suivant de recherche linéaire dans une liste.

```
>>> def est_present(t, x)
        # precondition {t: list}
        # postcondition {resultat = True si x est dans t, et False sinon}
        n = len(t)
        for k in range(n):
            if t[k] == x:
                return True
        return False
```

Pour prouver ce programme, nous allons définir les invariants

$$\mathcal{I}_k : \text{« } x \text{ n'est pas un des } t[0], \dots, t[k-1] \text{ »}$$

pour tout $k \in [\![0, n]\!]$. La seule différence avec les preuves précédentes est qu'il faut prouver que si $\mathcal{I}_k$ est vrai au debut du corps de la boucle et que le corps de la boucle s'exécute sans rencontrer de `break` ou de `return`, $\mathcal{I}_{k+1}$ est vrai en fin de boucle.

— $\mathcal{P} \Longrightarrow \mathcal{I}_0$: En effet, avant d'entrer dans la boucle, on peut affirmer que x ne fait pas partie de $\{t[i] : 0 \leqslant i < 0\}$ puisque cet ensemble est vide.

— $\mathcal{I}_k \longrightarrow \mathcal{I}_{k+1}$: Soit $k \in [\![0, n-1]\!]$. On suppose que x ne fait pas partie de l'ensemble $\{t[i] : 0 \leqslant i < k\}$. Si le programme ne sort pas de la boucle, il n'a pas rencontré de `return`, donc $t[k] \neq x$. On en déduit qu'en fin de boucle, on peut affirmer que x ne fait pas partie de l'ensemble $\{t[i] : 0 \leqslant i < k+1\}$.

— $\mathcal{I}_n \Longrightarrow \mathcal{Q}$: Si on arrive en fin de boucle, on a bien prouvé que x n'appartient pas à $\{t[i] : 0 \leqslant i < n\}$, c'est-à-dire à la liste. Dans ce cas, la fonction renvoie `False`, ce qui est correct. Si on sort de la fonction par le `return` au milieu de la boucle, c'est qu'on a trouvé $k \in [\![0, n-1]\!]$ tel que $t[k] = x$. On en déduit que x fait partie de la liste et donc que la réponse renvoyée doit être `True`.

On a donc prouvé la correction totale de l'algorithme de recherche linéaire.