

Spécification et preuve formelle de programmes : Introduction (TD1)

Spécification de programme

Afin de **prouver** qu'une fonction ^a est correcte, il est nécessaire de formaliser ce que la fonction est censée faire. Cette formalisation est appelée la **spécification** de la fonction. Il y a plusieurs façons de spécifier une fonction, ce cours se concentre sur la définition de contrats à l'aide de **préconditions** et de **postconditions**.

- La **précondition** permet de décrire dans quels cas la fonction est censée fonctionner correctement. Par exemple, si on considère la fonction `mod` de l'exercice précédent (la version de base), on pourrait dire que la précondition est " b ne vaut pas `UINT_MAX`". En effet, si cette condition n'est pas respectée, le comportement de la fonction est indéterminé, car elle effectue une division par zéro.
- La **postcondition**, quant à elle, permet de décrire ce que la fonction est censée faire **lorsque la précondition est respectée**. Pour la fonction `mod`, on pourrait dire que la postcondition est "la fonction renvoie le reste de la division de a par $b + 1$ ".

Une fois ces conditions définies, il reste à **prouver** que la fonction satisfait bien le contrat décrit par la précondition et la postcondition. Cependant, afin de réaliser des preuves rigoureuses, il est nécessaire de travailler avec des informations tout aussi rigoureuses. Ainsi, pour la fonction `mod`, si on utilise le contrat informel décrit précédemment, il est difficile de faire des preuves rigoureuses. Dans le reste de ce cours, on utilisera donc la logique du premier ordre pour formaliser précisément les préconditions et postconditions. Ci-dessous, on donne une version formelle de la précondition et de la postcondition de la fonction `mod`^b. Dans le reste du cours, on dénote la précondition d'une fonction par φ et la postcondition par ψ .

La précondition φ_{mod} de la fonction `mod` :

$$\varphi_{\text{mod}} : b \neq \text{UINT_MAX}$$

La postcondition ψ_{mod} de la fonction `mod` :

$$\psi_{\text{mod}} : \text{return} = a \% (b + 1)$$

a. Un programme, une suite d'instructions, une fonction, etc. Cette notion s'applique à tout sous-ensemble d'un code. Dans ce cours, ce sera appliquée à des fonctions.

b. On verra lors du prochain TD la syntaxe exacte que nous utiliserons, car elle dépend de Frama-C, l'outil de preuves de programmes que nous utiliserons.

Preuve de programme

Étant donné un programme P , une précondition φ et une postcondition ψ , on dénote par le **triple de Hoare** $\{\varphi\}P\{\psi\}$ la propriété suivante : "si le programme P est exécuté dans un état satisfaisant la précondition φ , alors, si P termine, il termine dans un état satisfaisant la postcondition ψ ".

Concrètement, un triplet de Hoare permet d'exprimer qu'un programme respecte une spécification décrite par une précondition et une postcondition (c'est donc juste une notation). Ainsi, pour prouver qu'un programme P est correct par rapport à une spécification (φ, ψ) , il suffit de prouver le triplet de Hoare $\{\varphi\}P\{\psi\}$. Afin de prouver des triplets de Hoare, on peut utiliser une technique qui consiste à calculer la **weakest precondition** (WP) du programme P par rapport à sa postcondition ψ . La weakest precondition, notée $\text{WP}(P, \psi)$, est la plus faible (moins restrictive) des préconditions telle que le triplet de Hoare $\{\text{WP}(P, \psi)\}P\{\psi\}$ soit vrai. Ainsi, si on peut calculer la weakest precondition d'un programme P par rapport à une postcondition ψ , alors pour prouver que le triplet de Hoare $\{\varphi\}P\{\psi\}$ est vrai pour une précondition φ , il suffit de prouver que $\varphi \implies \text{WP}(P, \psi)$. Idéalement, la précondition φ choisie est exactement la weakest precondition.

La weakest precondition peut être calculée de manière inductive en partant de la postcondition et en remontant le code. Pour ce faire, il faut dans un premier temps définir les règles de calcul de la weakest precondition pour les différentes instructions possibles^a. Voici quelques règles de calcul de la weakest precondition :

1. $\text{WP}(\text{skip}, \psi) \equiv \psi$
2. $\text{WP}(x := e, \psi) \equiv \psi[x \leftarrow e]$
3. $\text{WP}(*p = e, \psi) \equiv \psi[*p \leftarrow e]$

4. $WP(\mathbb{I}; \mathbb{I}', \psi) \equiv WP(\mathbb{I}, WP(\mathbb{I}', \psi))$
5. $WP(\text{if(cond)} \ \mathbb{I}; \text{else } \mathbb{I}', \psi) \equiv (\text{cond} \implies WP(\mathbb{I}, \psi)) \wedge (\neg \text{cond} \implies WP(\mathbb{I}', \psi))$
6. $WP(\text{return } e, \psi) \equiv \psi[\backslash \text{result} \leftarrow e]$

Où $\psi[x \leftarrow e]$ désigne la formule obtenue en remplaçant toutes les occurrences de la variable x par l'expression e dans la formule ψ . Par exemple, $WP(x = y + z, x > 20)$ indique qu'il faut remplacer toutes les occurrences de x par $y + z$ dans la formule $x > 20$, ce qui donne $y + z > 20$. Notons que dans la règle n°6, $\backslash \text{result}$ est une variable spéciale qui représente la valeur renournée par la fonction. On l'utilisera dans les postconditions pour faire référence à cette valeur.

```

1 int f(int a) {
2     int res = 0;
3     if(a > 2) {
4         res = a;
5     } else {
6         res = -a;
7     }
8     return res;
9 }
```

En utilisant ces règles, on peut calculer la weakest precondition d'une fonction par rapport à sa postcondition. Par exemple, considérons la fonction f ci-dessus, avec la postcondition :

$$\psi_f : \backslash \text{result} \geq 0$$

On peut calculer la weakest precondition de f par rapport à ψ_f comme suit :

$$\begin{aligned} & WP(2 - 8, \psi_f) \\ &= WP(2 - 8, \backslash \text{result} \geq 0)[\backslash \text{result} \leftarrow res] \\ &= WP(2 - 7, res \geq 0) \\ &= WP(2, (a > 2 \implies WP(4, res \geq 0)) \wedge (a \leq 2 \implies WP(6, res \geq 0))) \end{aligned}$$

On peut alors effectuer le calcul des deux sous-parties séparément :

$$\begin{aligned} & WP(4, res \geq 0)[res \leftarrow a] \\ &= a \geq 0 \end{aligned}$$

Et :

$$\begin{aligned} & WP(6, res \geq 0)[res \leftarrow -a] \\ &= -a \geq 0 \wedge a \neq INT_MIN \\ &= a \leq 0 \wedge a \neq INT_MIN \end{aligned}$$

Notons que dans le second calcul, on doit ajouter la condition $a \neq INT_MIN$ pour éviter un overflow lors de la négation de a . En effet, en C, $-a$ ne renvoie pas la bonne valeur si a vaut INT_MIN . On peut ensuite combiner les deux résultats dans l'équation originale :

$$\begin{aligned} & WP(2, (a > 2 \implies WP(4, res \geq 0)) \wedge (a \leq 2 \implies WP(6, res \geq 0))) \\ &= WP(2, (a > 2 \implies a \geq 0) \wedge (a \leq 2 \implies (a \leq 0 \wedge a \neq INT_MIN)))[res \leftarrow 0] \\ &= (a > 2 \implies a \geq 0) \wedge (a \leq 2 \implies (a \leq 0 \wedge a \neq INT_MIN)) \end{aligned}$$

Clairement, la première implication est toujours vraie. La deuxième peut être réécrite comme $a > 2 \vee (a \leq 0 \wedge a \neq INT_MIN)$. Donc la weakest precondition finale est : $WP(f, \psi_f) \equiv a > 2 \vee (a \leq 0 \wedge a \neq INT_MIN)$. Il faut cependant faire attention : Ce qui donne le triplet de Hoare :

$$\{a > 2 \vee (a \leq 0 \wedge a \neq INT_MIN)\}f\{\backslash \text{result} \geq 0\}$$

Autrement dit, la fonction renvoie un nombre positif ou nul si a est strictement plus grand que 2, égal à 0 ou négatif^b.

a. On ne fera pas de calculs sur des programmes trop complexes, donc on peut se permettre de restreindre la définition, le but est surtout de comprendre le fonctionnement général.

b. Je vous laisse vous convaincre que c'est effectivement vrai.

Exercice 1

```
1 unsigned int mod(unsigned int a, unsigned int b) {  
2     return a % (b + 1);  
3 }
```

1. Expliquer brièvement ce que fait la fonction ci-dessus.
2. La fonction contient un bug *a*, lequel ? Proposer une correction.
3. Comment s'assurer que la fonction est correcte ?
 - a. En tout cas, il y a de fortes chances que ce ne soit pas le comportement voulu.

Correction Exercice 1

1. La fonction renvoie le reste de la division de *a* par *b* + 1.

2. Le bug potentiel se trouve dans le fait que *b* + 1 peut provoquer un **overflow**. Cela se produit notamment lorsque *b* = `UINT_MAX`. En effet, `UINT_MAX` est le plus grand entier non signé qui peut être représenté en C, ainsi sa représentation binaire est 111...111 (le nombre de bits dépend de la machine). En ajoutant 1, on obtient alors 1000...000 (avec un bit de plus, le 1 tout à gauche), or ce bit ne peut pas être stocké, il est donc perdu. Ainsi, si on a *b* = `UINT_MAX`, on a *b* + 1 = 0 et donc on obtient une division par zéro dans la fonction.

Pour corriger ce bug, il y a plusieurs solutions, toutes discutables. Par exemple, on peut rajouter une condition pour vérifier que *b* n'est pas égal à `UINT_MAX` avant de faire le calcul. Il n'y a plus de risque d'overflow dans ce cas, mais il faut retourner une valeur, et il n'est pas clair quelle valeur retourner. En effet, si on se base uniquement sur un point de vue arithmétique, alors on a nécessairement *a* < *b* + 1 puisque *b* + 1 est plus grand que tout entier non signé représentable en C. Ainsi, on peut retourner *a*, comme dans la solution ci-dessous.

```
1 unsigned int mod(unsigned int a, unsigned int b) {  
2     if(b == UINT_MAX) {  
3         return a;  
4     }  
5     return a % (b + 1);  
6 }
```

Cependant, on peut aussi vouloir gérer ce cas d'erreur à part (par exemple, car un overflow n'est jamais censé se produire, donc on aimerait éviter une erreur silencieuse). Dans ce cas, une solution possible est de ne pas retourner la valeur, mais de la stocker dans un pointeur passé en argument. Ainsi, lorsque la valeur de *b* est trop grande, on peut retourner un code d'erreur et ne pas modifier le pointeur. Voici une solution possible :

```
1 // on pourrait faire plus propre avec une macro pour  
2 // le code d'erreur  
3 int mod(unsigned int a, unsigned int b, unsigned int * result) {  
4     if(b == UINT_MAX) {  
5         return -1;  
6     }  
7     *result = a % (b + 1);  
8     return 0;  
9 }
```

3. Pour s'assurer que la fonction est correcte, il y a deux grandes stratégies : la validation et la vérification. La validation consiste à tester la fonction sur un grand nombre de cas, en espérant que si elle passe tous les tests, alors elle est correcte. Le problème de cette stratégie est qu'elle n'est pas exhaustive, donc il se peut qu'il y ait des bugs, et dans des systèmes critiques, c'est peu souhaitable. La vérification, quant à elle, consiste à prouver mathématiquement que la fonction est **correcte**. Cependant, la notion de correction doit être précisée. Pour la fonction précédente, on voudrait exprimer une

propriété du type : "si b n'est pas égal à `UINT_MAX`, alors on stocke dans `*result` le reste de la division de a par $b + 1$, et on retourne 0, sinon on ne modifie pas `*result` et on retourne -1 ". Avec un exemple aussi simple, on peut supposer que cette **spécification** est claire et non ambiguë. Mais dans des programmes plus complexes, il peut être difficile de formuler précisément la spécification en langage naturel. Il est donc nécessaire de formaliser ce qu'est une spécification et comment la décrire de manière rigoureuse.

Une fois qu'on est tous accordés sur la spécification d'une fonction, il reste alors à **prouver** que la fonction respecte bien cette spécification.

Exercice 2

```

1 int max(int x, int y) {
2     int res;
3     if(x > y) {
4         res = x;
5     } else {
6         res = y;
7     }
8     return res;
9 }
```

1. Donner une spécification décrivant le comportement attendu de la fonction `max`.
2. Prouver que la postcondition proposée est bien respectée par la fonction `max` en calculant la weakest precondition.
3. La précondition proposée à la question 1. implique-t-elle la weakest precondition calculée à la question 2. ?

Correction Exercice 2

1. On peut donner une infinité de spécifications possibles pour la fonction `max`. Une spécification pertinente serait :

$$\varphi_{\max} : \top$$

$$\psi_{\max} : (x > y \implies \text{\textbackslash result} = x) \wedge (x \leq y \implies \text{\textbackslash result} = y)$$

2. On veut prouver que le triplet de Hoare $\{\varphi_{\max}\} \max \{\psi_{\max}\}$ est vrai. Pour ce faire, il faut prouver que la weakest precondition de la fonction `max` par rapport à la postcondition ψ_{\max} est impliquée par la précondition φ_{\max} .

On calcule donc la weakest precondition :

$$\begin{aligned}
 & WP(2 - 8, (x > y \implies \text{\textbackslash result} = x) \wedge (x \leq y \implies \text{\textbackslash result} = y))[\text{\textbackslash result} \leftarrow \text{res}] \\
 &= WP(2 - 7, (x > y \implies \text{res} = x) \wedge (x \leq y \implies \text{res} = y)) \\
 &= WP(2, (x > y \implies WP(4, (x > y \implies \text{res} = x) \wedge (x \leq y \implies \text{res} = y))) \wedge \\
 &\quad (x \leq y \implies WP(6, (x > y \implies \text{res} = x) \wedge (x \leq y \implies \text{res} = y))))
 \end{aligned}$$

On peut traiter les deux WP séparément. D'une part, on a :

$$\begin{aligned}
 & WP(4, (x > y \implies \text{res} = x) \wedge (x \leq y \implies \text{res} = y))[res \leftarrow x] \\
 &= (x > y \implies x = x) \wedge (x \leq y \implies x = y)
 \end{aligned}$$

D'autre part, on a :

$$\begin{aligned}
 & WP(6, (x > y \implies \text{res} = x) \wedge (x \leq y \implies \text{res} = y))[res \leftarrow y] \\
 &= (x > y \implies y = x) \wedge (x \leq y \implies y = y)
 \end{aligned}$$

On peut remplacer dans l'expression initiale :

$$\begin{aligned}
 & WP(2, (x > y \implies (x > y \implies x = x) \wedge (x \leq y \implies x = y)) \wedge \\
 &\quad (x \leq y \implies (x > y \implies y = x) \wedge (x \leq y \implies y = y))) \\
 &= (x > y \implies (x > y \implies x = x) \wedge (x \leq y \implies x = y)) \wedge \\
 &\quad (x \leq y \implies (x > y \implies y = x) \wedge (x \leq y \implies y = y))
 \end{aligned}$$

La dernière ligne est donc notre weakest precondition. On peut cependant la simplifier. La première ligne nous dit que si $x > y$, alors *si* $x > y$, *on a* $x = x$. Clairement, $x = x$ est toujours vrai, donc on a en fait

$$x > y \implies (x > y \implies \top)$$

De plus, on suppose que $x > y$ avant l'implication, donc nécessairement, dans la partie droite, $x > y$ est vrai. Cela donne :

$$x > y \implies \top$$

Et cette expression est toujours vrai, c'est évident si on réécrit l'implication sous forme de disjonction :

$$\neg(x \leq y) \vee \top$$

De même, la première ligne nous dit aussi que si $x > y$, alors *si* $x \leq y$, *on a* $x = y$. Bon, clairement, puisqu'on suppose que $x > y$, on a nécessairement $x \leq y$ faux, c'est à dire :

$$x > y \implies (\perp \implies x = y)$$

Or, une implication avec une partie gauche fausse est toujours vraie, ce qui se voit en réécrivant l'implication sous forme de disjonction :

$$\top \vee (x = y)$$

On a donc à nouveau une implication dont la partie droite est toujours vraie :

$$x > y \implies \top$$

En utilisant le même raisonnement que précédemment, on a donc la première ligne qui est toujours vraie. On peut faire le même raisonnement pour la deuxième ligne (si tout cela n'est pas très clair, je vous invite fortement à faire la preuve de votre côté), ainsi, la deuxième ligne est aussi toujours vraie. La weakest precondition se simplifie donc en $\top \wedge \top$, c'est-à-dire \top . Et c'est une bonne chose, puisque la précondition qu'on avait choisie était \top , et clairement, on a $\top \implies \top$ qui est vraie. Donc notre précondition implique bien la weakest precondition, qui elle-même implique la postcondition. Donc le triplet de Hoare $\{\varphi_{\max}\} \max \{\psi_{\max}\}$ est vrai.

3. Oui :)^a

a. En même temps, c'est moi qui écrit les exercices, encore heureux ! Mais peut-être que ce n'était pas votre cas (et ce n'est pas grave si c'est le cas, l'important, c'est de s'en rendre compte).

Exercice 3

```
1 void swap(int * a, int * b) {
2     int tmp = *a;
3     *a = *b;
4     *b = tmp;
5 }
```

1. Donner une spécification décrivant le comportement attendu de la fonction `swap`.
2. Prouver que la postcondition proposée est bien respectée par la fonction `swap` en calculant la weakest precondition.
3. La précondition proposée à la question 1. implique-t-elle la weakest precondition calculée à la question 2. ?

Exercice 4

```
1 int div(int a, int b) {
2     int res = 0;
3     if(a == 0) {
4         res = 1;
5     } else {
6         res = b / a;
7     }
8     return res;
9 }
```

1. Donner une spécification décrivant le comportement attendu de la fonction `div`.
2. Prouver que la postcondition proposée est bien respectée par la fonction `div` en calculant la weakest precondition.
3. La précondition proposée à la question 1. implique-t-elle la weakest precondition calculée à la question 2.?

Exercice 5

```
1 int sum(int n) {
2     int res = 0;
3     int i = 0;
4     while(i <= n) {
5         res = res + i;
6         i = i + 1;
7     }
8     return res;
9 }
```

Donner une spécification décrivant le comportement attendu de la fonction `sum`, qui fait la somme des entiers de 0 à n .

Exercice 6

```
1 int is_sorted(int * tab, unsigned int size) {
2     unsigned int i = 1;
3     int res = 1;
4     while(i < size) {
5         if(tab[i - 1] > tab[i]) {
6             res = 0;
7         }
8         i++;
9     }
10    return res;
11 }
```

1. Donner une spécification décrivant le comportement attendu de la fonction `is_sorted`, qui renvoie 1 si le tableau est trié en ordre croissant, et 0 sinon.