

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. $WP(\text{if}(\text{cond}) \mathbb{I}; \text{else } \mathbb{I}', \psi) \equiv (\text{cond} \Rightarrow WP(\mathbb{I}, \psi)) \wedge (\neg \text{cond} \Rightarrow WP(\mathbb{I}', \psi))$
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 :

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

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 :

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

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.

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. ?

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.