Conception Formelle

2018-2019

TP noté: Quelques algorithmes de tableaux.

Vincent Penelle

- Faites des brouillons, et, une fois les deux premières heures passées, aidez-vous de frama-c.
- Sur papier, les définition des prédicats/fonctions peuvent être donnés en logique naturelle (mais en précisant les labels lorsque nécessaire). On ne demande pas de justification particulière des prédicats.
- Sur papier, annoncez clairement les invariants/variants de boucles avant de prouver chaque partie de la preuve séparément (après/pendant/avant la boucle). Les preuves de logiques du premier ordre pourront être éludées (mais justes), mais pas les calculs de WP. On pourra cependant éluder les calculs de WP identiques (ou presque) à des calculs précédents.
- Pour les deux derniers exercices, la preuve sur papier est facultative, mais appréciée (elle est aussi a priori plus longue).
- On préfèrera des preuves incomplètes sur papier et un code partiellement annoté à pas de rendu papier du tout et des codes entièrement annotés.
- On préfèrera peu d'exercices bien fait que tous mal faits.
- Deadline : mercredi 12h15 (copie papier + code moodle).

Exercice 1: Swap On considère la fonction `swapArray` suivante.

```
void swapArray(int *t, int i, int n, int a, int b){
1. if(a<i || a>=n || b<i || b>=n) return;
2. else{
3.       int aux = t[a];
4.       t[a] = t[b];
5.       t[b] = aux;
6.}}
```

1. Donnez un prédicat `unchangedTab{L,M}(int *tab, int *tab2, integer i, integer n)` qui est vrai si et seulement le `tab` et `tab2` sont identiques entre i et n-1 inclus (`tab` étant évalué au label L, `tab2` au label M).
2. Donnez un prédicat `wrongRangedSwap{L,M}(int *t, int *s, integer i, integer n, integer j, integer k)` qui décrit le comportement de la fonction swap quand j et k ne sont pas des indices permettant de faire un swap (i.e., en dehors des bornes i et n du tableau). Dans ce cas, swap ne fait rien (i.e., t et s sont identiques).
3. Donnez un prédicat `swapInArrayAux{L,M}(int *t, int *s, integer i, integer n, integer j, integer k)` qui décrit le comportement de la fonction swap lorsque $i \leq j \leq k < n$.
4. En utilisant les deux prédicats précédent, donnez un prédicat `swapInArray{L,M}(int *t, int *s, integer i, integer n, integer j, integer k)` qui décrit le comportement de la fonction swap quelques soient les valeurs de i, n, j et k.

5. Démontrez (sur papier) que le code donné vérifie les post-conditions `/*@ ensures swapInArrayAux {Pre,Post} (t,t,i,n,j,k)*/`, et `/*@ ensures wrongRangedSwap {Pre,Post} (t,t,i,n,j,k)*/`. Vous aurez besoin de manipulez l'expression logique du prédicat (mais contenant unchangedTab). **On admettra que WP(x=t,Pred{L}) = Pred{L'} si x n'est pas une variable apparaissant dans Pred (L et L' correspondant au label après et avant l'instruction x=t).** Vous garderez le prédicat unchangedTab tel quel dans les preuves (et `unchangedTab{L,L}(t,t,i,n)` est vrai). Pourquoi est-ce suffisant pour prouver le contrat total ?
6. Vérifiez que frama-c arrive à démontrer `/*@ ensures swapInArray {Pre,Post} (t,s,i,n,j,k)*/`, et ajoutez la clause `assigns` vérifiée par cette fonction, ainsi que la clause requires nécessaires pour valider les gardes RTE.

Exercice 2: Minimum On considère la fonction minimum.

```
int minimum(int *t, int i, int n){
1.  if(n<=i) return n-1;
2.  else{
3.        int min = i;
4.        int k = i+1;
5.        while(k<n){
6.                if(t[k]<t[min]) min = k;
7.                k++;
8.        }
9.        return min;
10.}}
```

1. Donnez un prédicat `minimum(int *t, integer i, integer n, integer min)` qui est vrai si et seulement si `min` est l'indice minimum du tableau `t` entre les indices `i` et `n-1` (inclus).
2. Prouvez (sur papier) que le code donné vérifie la post-condition `/*@ ensures i<n ==> minimum(t,i,n,\result)*/`. Vous aurez besoin de deux invariants de boucles, un parlant de k (et permettant de connaître la valeur de k à la fin de la boucle), et un permettant de démontrer la post-condition. Vous devrez (dans la preuve de l'invariant correspondant) développer l'expression de minimum, mais pas trop.
3. Démontrez (sur papier) également que cette fonction termine (i.e., trouvez un variant de boucle et prouvez-le).
4. Sur frama-c, donnez et prouvez deux post-conditions (la précédente, plus une pour n<=i), et donnez la clause assigns pertinente, ainsi que les éventuelles préconditions permettant de valider les gardes RTE. N'oubliez pas que vous aurez besoin d'une clause `loop assigns`. Cette spécification pourra (ou non) faire intervenir des comportements.

Exercice 3: countTab On considère la fonction suivante renvoyant le nombre d'éléments d'un tableau égaux à une valeur donnée.

```
int countTab(int *t, int n, int val){
1.  if(n<=0) return 0;
2.  else{
3.        int res = 0;
4.        int i = 0;
```

```
5.        while(i<n){
6.                if(t[i] == val) res++;
7.                i++;
8.        }
9.        return res;
10.}}
```

1. On va d'abord définir, via des axiome, la fonction `logic integer countTab(int *t, integer i, integer n, int v) reads t[i .. n-1]`. Donnez un axiome disant que, pour tout tableau t, tout i, tout n et tout v, si le tableau est vide (i.e., $n <= i$), alors le tableau ne contient aucun v.
2. Donnez un axiome disant que, pour tout tableau t, et entiers i,n, et v, si on connaît `countTab(t,i,n-1,v)` et `t[n-1] == v`, la valeur de `countTab(t,i,n,v)` est ...
3. Donnez un axiome disant que, pour tout tableau t, et entiers i,n, et v, si on connaît `countTab(t,i,n-1,v)` et `t[n-1] != v`, la valeur de `countTab(t,i,n,v)` est ...
4. Démontrez (sur papier) que le code fourni calcule bien la fonction définie par les axiomes précédents (i.e., `/*@ ensures \result == countTab(t,0,n,val);*/`. Vous aurez besoin de deux invariants de boucles, un parlant de i (et permettant de connaître la valeur de i à la fin de la boucle), et un permettant de démontrer la post-condition.
5. Démontrez (sur papier) la terminaison de la fonction.
6. Sur frama-c, prouvez la post-condition précédente, et donnez la clause assigns pertinente, ainsi que les éventuelles préconditions permettant de valider les gardes RTE (pour cela, vous aurez besoin d'un invariant de boucle supplémentaire – il n'est pas demandé de prouver ce dernier sur papier). N'oubliez pas que vous aurez besoin d'une clause `loop assigns`.

Exercice 4: countMinimum (un peu plus dur) La fonction suivante calcule le nombre d'occurrence de l'élément minimal d'un tableau.

```
int countMinimum(int *t, int i, int n){
1. if(n<=i) return 0;
2. else{
3.        int min = i;
4.        int count = 1;
5.        int k = i+1;
6.        while(k<n){
7.                if(t[k] < t[min]){
8.                        min = k;
9.                        count = 0;
10.               }
11.               if(t[k] == t[min]) count++;
12.           k++;
13.       }
14.       return count;
15.}}
```

1. Démontrez que le code ci-dessus vérifie `/*@ ensures n<=i ==> \result == 0;*/`.

2. Écrivez une post-condition décrivant la post-condition de cette fonction quand le tableau n'est pas vide (le résultat est égal à `countTab(t,i,n,val)` pour val tel que `minimum(t,i,n,val)` – vous aurez besoin d'un quantificateur existentiel.).
3. Démontrez que le code ci-dessus vérifie la post-condition précédente, dans le cas où $i \leq n$. Vous aurez besoin 4 invariants de boucles. 3 comme d'habitude (un parlant de k et permettant de déduire la valeur de k en fin de boucle, un parlant de min et un parlant de la valeur de min) qui permettent de déduire la post-condition. Vous aurez également besoin d'un invariant supplémentaire disant que pour toute valeur plus petite que `t[min]`, le nombre d'élément dans le tableau de cette valeur est nul.
4. Démontrez la terminaison de la fonction.
5. Ajoutez la clause assigns pertinente, ainsi que les éventuelles préconditions permettant de valider les gardes RTE (pour cela, vous aurez besoin d'un invariant de boucle supplémentaire). N'oubliez pas que vous aurez besoin d'une clause `loop assigns`.

Exercice 5: sortByPermutation (plus dur) On met maintenant un peu toutes les fonctions précédentes ensembles pour un algorithme de tri.

Attention, ici, pour les preuves sur papier, vous aurez à remonter deux appels de fonctions. Pour les démontrer, vous découperez chaque formule en deux parties : celle parlant de valeurs modifiées par la fonction (dans le assigns), et celle parlant de valeurs invariantes. La deuxième partie devra être vraie avant l'appel de la fonction, tandis que la première devra être impliquée par la post-condition de la fonction (et vous aurez donc à démontrer que la pré-condition est vraie avant l'appel – mais ici, les pré-conditions sont $\top$, donc ça devrait aller).

```
void sortByPermutation(int *t, int i, int n){
1. if(n<i) return;
2. else{
3.       int j = i;
4.       while(j<n){
5.               int min = minimum(t,j,n);
6.               swapArray(t,i,n,j,min);
7.               j++;
8.}}}
```

1. Donnez un prédicat `predicate sorted(int *t, integer i, integer n)` qui est vrai si et seulement si le tableau `t` est trié entre i et n.
2. Démontrez que le code ci-dessus vérifie la post-condition `/*@ ensures sorted{Post}(t,i,n);*/`. Pour cela, vous aurez besoin de 3 invariants de boucles : un parlant de j, un parlant de la partie déjà triée du tableau (comme d'habitude), mais vous aurez aussi besoin d'un invariant indiquant que le dernier élément que la partie triée du tableau est plus petit que tous les éléments situés après lui (attention, au début de la boucle, cet élément n'existant pas, il faut, grâce à une implication bien choisie, assurer que l'invariant reste vrai).
3. Donnez un prédicat `permutation{L,M}(int *t, int *s, integer i, integer n) reads \at(t[i .. n-1],L),\at(s[i .. n-1],M);` qui décrit que le tableau s est une permutation du tableau t. Vous aurez besoin de deux axiomes pour ceci : un disant qu'un tableau est une permutation de lui-même ; et un disant que si on a trois tableaux t,s et u, si s est une permutation de t et qu'il existe j et k tels que `swapInArray(s,u,i,n,j,n)`, alors u est une permutation de t.

4. Démontrez que le code ci-dessus vérifie la post-condition `/*@ ensures permutation {Pre,Post}(t,t,i,n);*/`. Vous aurez besoin d'un invariant de boucle supplémentaire.
5. Ajoutez la clause assigns pertinente, ainsi que les éventuelles préconditions permettant de valider les gardes RTE. N'oubliez pas que vous aurez besoin d'une clause `loop assigns`.