Preuve de programme par Frama-C et WP

Approches formelles pour la vérification de programmes, Master CNS

2020–21

Le but de cette séance machine est d'utiliser le plugin WP de l'outil Frama-C [1], qui utilise la logique de Hoare, et plus précisément le calcul de précondition la plus faible pour permettre de prouver des spécifications de programmes annotés écrits en C.

1 Échange

1. Dans un fichier `swap1.c`, écrire la fonction C annotée suivante :

   ```
   void swap(int a, int b) {
     int x = a;
     int y = b;
     int tmp;
     //@ assert x == a && y == b;
     tmp = x;
     x = y;
     y = tmp;
     //@ assert y == a && x == b;
   }
   ```

   Les commentaires commençant par `@` sont utilisés par le plugin WP pour indiquer les annotations du programme. En particulier, les `//@ assert ...` permettent d'ajouter les assertions des triplets de Hoare.

2. Exécuter `frama-c-gui swap1.c`. Cliquer sur `swap1.c` à gauche, son contenu apparaît. En faisant un clic droit sur les `assert`, demander au plugin WP de prouver les assertions.

3. Ouvrir l'onglet "WP goals" en bas. On constate deux ronds verts dans la colonne Qed, ce qui veut dire que les assertions ont été prouvées uniquement en simplifiant les conditions de vérification, sans faire appel à un prouveur extérieur.

4. Fermer Frame-C, et écrire un nouveau fichier `swap2.c` :

   ```
   void swap(int a, int b) {
     int x = a;
     int y = b;
     //@ assert x == a && y == b;
     x = 2 * (x + y);
   ```

1. `http://frama-c.com/`

```
  y = x / 2 - y;
  x = x / 2 - y;
  //@ assert y == a && x == b;
}
```

Relancer Frama-C, prouver les assertions. Dans l'onglet "WP proof obligations", on remarque que la deuxième assertion a nécessité d'utiliser un prouveur externe, en l'occurence Alt-Ergo. Alt-Ergo [2] est un prouveur automatique de la famille des prouveurs SMT, qui sont particulièrement bons en arithmétique linéaire (i.e. sans produit de variables) et quand il n'y a pas de quantificateurs.

5. En général, on n'écrit pas les assertions, mais on écrit les spécifications des fonctions du programme. Écrire un fichier `swap3.c` contenant :

```
/*@ ensures A: *a == \old(*b) ;
  @ ensures B: *b == \old(*a) ;   */
void swap(int *a,int *b)
{
  int tmp = *a ;
  *a = *b ;
  *b = tmp ;
}
```

L'annotation `ensures ...` indique le contrat que doit respecter la fonction ; `\old(x)` signifie la valeur qu'avait `x` avant l'exécution de la fonction.

Utiliser Frama-C pour prouver que la fonction vérifie bien sa spécification.

2 Somme d'impairs

On veut montrer que la somme des n premiers nombres impairs est égale à n^2. Dans un fichier `sum.c`, on considère la fonction :

```
/*@ ensures \result == n * n; */
int f(int n) {
  int i = 0;
  int s = 0;
  while (i != n) {
    i++;
    s += 2 * i - 1;
  };
  return s;
}
```

Comme son nom l'indique, `\result` correspond au résultat retourné par la fonction.

6. Dans Frama-C, constater qu'il n'est pas possible de prouver la correction de la fonction.
7. Pour pouvoir montrer cette correction, il faut ajouter une annotation à la boucle, notamment en indiquant l'invariant de boucle. On doit aussi indiquer quelles variables

2. `http://alt-ergo.lri.fr/`

sont modifiées dans le corps de la boucle. Pour cela, on ajoute juste avant la boucle les annotations suivantes :

```
/*@ loop invariant i * i == s;
  @ loop assigns i, s; */
```

Relancer Frama-C, on peut maintenant prouver l'assignation et la correction de la fonction et de l'invariant.

3 Exemples du cours

Donner la spécification, écrire l'implémentation, puis montrer la correction des fonctions suivantes :

8. `int max(int a, int b)` qui renvoie le max de deux entiers ;
9. `indice_max(int t[], int n)` qui renvoie l'indice de la valeur maximum du tableau `t` de taille `n` ;
10. `array_max(int t[], int n)` qui renvoie la valeur maximum du tableau `t` de taille `n` ;

4 Reste de la division euclidienne

On considère la fonction suivante, qui doit retourner le reste de la division euclidienne de `a` par `b`

```
int rem(int a, int b) {
  while (a >= b)
    a = a - b;
  return a;
}
```

11. Donner la spécification de `rem` ; Pour faciliter la preuve, utiliser `integer` et non `int` comme type pour le quotient ;
12. Trouver un invariant de boucle ;
13. Montrer que le contrat de la fonction est valide en supposant que l'invariant est vérifié ;
14. Essayer de démontrer l'invariant de boucle. Alt-ergo n'y arrive pas. Nous allons utiliser Coq pour arriver à nos fins.

    L'utilisation de Coq avec Frama-C est dépréciée, mais il est possible de l'activer en passant l'option `-wp-prover=native:coq` à `frama-c-gui`.

    Dans l'onglet "WP Goals", on constate qu'Alt-Ergo a échoué pour la preuve de la préservation de l'invariant. En regardant la formule à démontrer, cela n'est pas surprenant puisqu'il y a un quantificateur existentiel. Nous allons donc utiliser Coq pour démontrer ce sous-but, en double-cliquant dans la case de la colonne Coq et de la ligne Invariant (established).

    Une fenêtre de l'environnement graphique de preuve Coq s'ouvre.

5 Factorielle

On souhaite maintenant montrer la correction de la fonction factorielle impérative (que l'on mettra dans un fichier `fact.c`) :

```
int fact(int n) {
  int y = 1;
  int x = n;
  while (x != 1) {
    y = y * x;
    x = x - 1;
  };
  return y;
}
```

15. Il faut être capable d'exprimer la spécification de la fonction `fact`, donc pour cela il faut que l'on dispose de la définition (au sens mathématique) de la factorielle. Pour cela, on peut ajouter une axiomatique au fichier :

    ```
    /*@ axiomatic Fact {
      @   logic integer Fact(integer n);
      @   axiom Fact_1: Fact(1) == 1;
      @   axiom Fact_rec: \forall integer n; n > 1 ==> Fact(n) == n * Fact(n-1);
      @ } */
    ```

    On déclare une fonction logique `Fact`, et on décrit à l'aide d'axiomes comment elle est définie.

16. On peut ensuite définir la spécification de `fact` :

    ```
    /*@ requires n > 0;
      @ ensures \result == Fact(n); */
    ```

    Ici, `requires` indique que le contrat ne vaut que si l'entier passé en argument est strictement positif.

17. Pour montrer la spécification, il faut rajouter un invariant à la boucle (et aussi indiquer que seules `x` et `y` sont modifiées). On rappelle qu'on a vu dans le cours qu'un invariant utile est :

    $$y * x! = n! \ \wedge \ n > 0 \ \wedge \ x > 0$$

    Montrer la correction de la fonction.