Spécification et preuve formelle de programmes : ACSL (TD3)

Variant et behaviors

Frama-C est un outil puissant contenant de nombreuses fonctionnalités. Nous n'avons vu ici qu'une petite partie de ses capacités, notamment les modules WP et RTE. Et même parmi ces modules, nous n'avons vu qu'une partie des possibilités offertes. Il reste notamment de nombreuses annotations ACSL que nous n'avons pas abordées. Par exemple, à mon grand dam, la plupart de vos versions Frama-C ajoutent automatiquement une clause vis-à-vis de la terminaison des fonctions (`\terminates`). Et c'est en effet un aspect important de la correction d'un programme. Ainsi, pour prouver qu'une fonction termine, il est nécessaire, en particulier, de prouver que toutes les boucles qu'elle contient terminent. Pour ce faire, on utilise des **variants de boucle**. Un variant de boucle est une expression entière qui doit être :

- Toujours positive au début de chaque itération de la boucle ;
- Décroissante à chaque itération de la boucle.

S'il est possible de prouver l'existence d'un tel variant de boucle, alors on peut naturellement conclure que la boucle termine, puisque le variant possède un nombre fini de valeurs possibles. Ainsi, dans l'exemple suivant, on peut utiliser le variant de boucle `n - i` pour prouver que la boucle termine. En ajoutant l'annotation `\terminates` à la fonction, on peut tester avec Frama-C, et vérifier que la preuve de terminaison passe correctement.

```
1 /*@ terminates \true; */
2 int sum(unsigned int n) {
3     unsigned int res = 0;
4     unsigned int i = 0;
5     /*@ loop variant n - i */
6     while(i < n) {
7         res += i;
8         i++;
9     }
10    return res;
11 }
12
```

Un autre outil très utile est la possibilité de définir des **behaviors** (comportements) pour une fonction. Il arrive souvent qu'en fonction des arguments passés à une fonction, le comportement de cette fonction change. Par exemple, dans le code ci-dessous, la fonction effectue soit une addition, soit une soustraction en fonction de la valeur de `a`.

```
1 Status perform_action(Action a, int x, int y, int * res) {
2     switch(a) {
3         case ADD: if((y >= 0 && x > INT_MAX - y) || (y <= 0 && x < INT_MIN - y)) return FAIL;
4                   else { *res = x + y; return SUCCESS; }
5         case SUB: if((y >= 0 && x < INT_MIN + y) || (y <= 0 && x > INT_MAX + y)) return FAIL;
6                   else { *res = x - y; return SUCCESS; }
7         default: return FAIL;
8     }
9     return FAIL;
10 }
11
```

On *pourrait* écrire une spécification ACSL pour cette fonction avec les outils vus jusqu'à présent, mais cela serait fastidieux et peu lisible. On peut alors utiliser des behaviors afin de décrire le comportement de la fonction en fonction de la valeur de ses arguments, par exemple de `a`. Un exemple est donné dans le fichier `TD3/src/behaviors_example.c`. On remarque que chaque behavior est constituée d'une précondition décrivant le cas où ce comportement s'applique, par exemple `assumes a == ADD;` pour le comportement d'addition. De plus chaque behavior possède aussi une post-condition décrivant l'effet de la fonction dans ce cas. Enfin, notons les deux dernières lignes `complete behaviors`

`add, sub, fail;` et `disjoint behaviors add, sub, fail;`. Si la première est prouvée, cela indique que les comportements listés couvrent tous les cas possibles (c'est-à-dire que pour toute valeur des arguments, au moins un des comportements s'applique). Si la seconde est prouvée, cela indique que les comportements listés sont mutuellement exclusifs (c'est-à-dire que pour toute valeur des arguments, au plus un des comportements s'applique). Lorsqu'on définit des behaviors, il est préférable de vérifier que ces deux propriétés sont vraies, afin d'éviter des comportements non couverts ou des comportements qui se chevauchent.

Il y a une différence fondamentale entre le mot clé `assumes` et le mot clé `requires` vu précédemment (qui peut aussi apparaître dans un behavior). En effet, une précondition `requires P;` indique que la fonction ne doit être appelée que si la propriété `P` est vraie. Tandis qu'une précondition `assumes P;` indique que le comportement décrit par le behavior s'applique uniquement lorsque la propriété `P` est vraie. Ainsi, si on utilise le mot clé `assumes`, on exprime le comportement de la fonction si la propriété est vraie, mais on ne dit pas implicitement qu'il n'existe pas un autre comportement si la propriété est fausse. Tandis qu'avec le mot clé `requires`, on exprime que la fonction ne doit pas être appelée si la propriété est fausse, et donc qu'il n'existe pas d'autre comportement dans ce cas. On peut observer cette différence dans le fichier `TD3/src/requires_vs_assumes.c`. En analysant ce fichier avec Frama-C, on remarque que la preuve de la fonction `div_2` échoue, cela vient du fait que du point de vue de Frama-C, il existe des cas non considérés (par exemple lorsque `y == 0`).

Prédicats et fonctions logiques

Comme dans tout langage de programmation, il est possible de définir des sortes de fonctions ou variables, qui permettent d'éviter la redondance. Par exemple, il est possible de définir des **prédicats** en ACSL, qui sont des formules logiques nommées, que l'on peut réutiliser dans les annotations ACSL. Par exemple, le prédicat suivant prend en entrée deux entiers `x` et `y`, et renvoie `true` si la division de `x` par `y` ne cause pas d'overflow (ou d'underflow) :

```
/*@ predicate valid_div(integer a, integer b) =
        (a != INT_MIN || b != -1) && b != 0;
*/

/*@ requires valid_div(x, y);
    ensures \result == x / y;
*/
int div(int x, int y) {
    return x / y;
}

```

Cet exemple peut être testé dans le fichier `TD3/src/predicate_example.c`. Il est aussi possible de donner des labels aux prédicats, afin d'exprimer la temporalité. Par exemple, le prédicat ci-dessous permet de comparer la valeur de $*x$ à moment L et à moment M.

```
/*@ predicate cmp{L, M}(int * x) =
        \at(*x, L)  == \at(*x, M);
*/
```

Quelques valeurs possibles de L et M sont[a] :

- `Pre` : avant l'exécution de la fonction ;
- `Here` : à l'endroit courant dans le code ;
- `Post` : après l'exécution de la fonction.

Ainsi, si on utilise le prédicat `cmp{Pre, Post}(x)` dans la spécification d'une fonction, on exprime que la valeur pointée par `x` ne change pas durant l'exécution de la fonction. Un exemple est donné dans le fichier `TD3/src/temporal_predicate_example.c`.

De la même manière, il est possible de définir des **fonctions logiques** en ACSL, qui sont des fonctions nommées que l'on peut réutiliser dans les annotations ACSL. Par exemple, la fonction logique suivante calcule la valeur absolue d'un entier :

```c
/*@ logic integer abs(integer x) =
      (x >= 0 ? x : -x);
*/

/*@ requires x != INT_MIN;
    ensures \result == abs(x);
*/
int abs(int x) {
    if(x >= 0) return x;
    return -x;
}

```

Cet exemple est disponible dans le fichier `TD3/src/logic_example.c`.

a. Il en existe quelques autres, mais dans le cadre de ce cours, elles ne sont pas nécessaires.

Exercice 1

Compléter le fichier `TD3/src/ex1.c` afin que Frama-C puisse prouver la spécification que vous aurez écrite.

Une fois que Frama-C est en mesure de prouver votre spécification, utiliser exactement la même spécification dans le fichier `ex1_bonus.c`, et assurez-vous de ne rien avoir oublié.

Exercice 2

Donner une spécification à la fonction dans `TD3/src/ex2.c`. Il est probable que dans les invariants de boucle, vous souhaitiez parler de la valeur des cases du tableau `a` avant la boucle. Pour ce faire, vous pouvez utiliser le mot clé `\at(..., Pre)` (car `\old` n'est pas défini en dehors de la spécification d'une fonction).

De plus, il est possible que les tableaux se chevauchent (d'un point de vue théorique, en pratique, on est bien d'accord que ce n'est pas une bonne idée de faire ça). Cependant, pour Frama-C cela est tout à fait possible et pour faire une preuve complète, il est nécessaire de le prendre en compte. Pour spécifier que les tableaux ne se chevauchent pas, vous pouvez utiliser l'annotation `\separated(...)`. Par exemple, pour indiquer que les tableaux `a` et `b` de taille `n` et `m` respectivement ne se chevauchent pas, vous pouvez écrire :

```
\separated(a + (0 .. n - 1), b + (0 .. m - 1))
```

Enfin, une autre notation qui peut vous être utile est celle permettant de désigner une plage d'indices dans un tableau (ce sera utile pour indiquer quelles parties du tableau sont modifiées par la boucle) :

```
loop assigns a[0 .. n - 1]
```

Exercice 3

Compléter le fichier `TD3/src/ex3.c` afin que Frama-C puisse prouver la spécification que vous aurez écrite.

Une fois cela fait, essayer d'utiliser votre même spécification dans le fichier `ex3_bonus.c`. Il y a de fortes chances que tout soit validé, pourtant la fonction `sort` ne trie pas correctement le tableau (il réécrit la première case).

Exercice 4

Un problème courant avec les expressions logiques est qu'elles sont difficiles à utiliser dans des preuves pour Frama-C, en particulier lorsqu'elles sont récursives. Pour observer ce phénomène, analyser le fichier `TD3/src/sum_example.c` avec Frama-C. Constatez que l'analyse n'est pas concluante (je vous laisse vous convaincre à votre rythme qu'elle devrait l'être).

Le problème vient du fait que les expressions logiques récursives sont difficiles à manipuler pour les prouveurs automatiques. En effet, pour utiliser une telle expression, le prouveur doit la dérouler totalement (ce qui est en général infaisable pour des entrées de taille non bornée). Pour résoudre ce problème, on peut utiliser des **axiomes** pour définir le comportement d'une expression logique sans expliquer comment la calculer. Cela permet aux prouveurs automatiques d'utiliser

directement les propriétés de l'expression logique, sans avoir à la dérouler. Dans le même fichier, essayez d'enlever l'espace avant l'arobase à la ligne `axiomatic SumTab ...` (et pensez à ajouter un espace avant la définition de la fonction logique `partial_sum` qui est juste au-dessus). En relançant l'analyse, vous verrez que Frama-C arrive à prouver la fonction `sum` correctement.

Les axiomes sont un outil puissant mais :

- Il n'est pas facile de savoir quels axiomes sont utiles ;
- Les axiomes sont des propriétés non prouvées (comme un vrai axiome en mathématique), ainsi si vous écrivez n'importe quoi, Frama-C pourra le réutiliser, ce qui peut mener à des preuves incorrectes.

Par exemple, l'axiome `SumRec` exprime le fait que la somme des valeurs entre les indices `start` et `end` est égale à la somme des valeurs entre `start` et `end - 1`, plus la valeur à l'indice `end - 1`. Concrètement, si `t = [1, 2, 3, 4, 5]`, `start = 1` et `end = 3`, ce que dit l'axiome, c'est que la somme `t[1] + t[2]` est égale à `t[1] + t[2] + t[3] - t[3]`. Clairement, cette propriété est vraie, mais on aurait pu écrire :

$$partial\_sum(t, start, end) == partial\_sum(t, start, end + 1)$$

et là clairement, ce n'est pas vrai (sauf si `t[end] = 0`). Pourtant, Frama-C aurait pu utiliser cet axiome incorrect pour prouver des propriétés fausses [a].

La vraie question est alors : comment savoir quels axiomes sont utiles (et idéalement corrects) ? En général, c'est assez difficile, mais une intuition que vous pouvez avoir est que les axiomes sont utiles pour représenter des propriétés récursives. Et ces dernières sont utiles dans les boucles. Par exemple, toujours dans le même fichier, on a l'invariant de boucle :

```
result == partial_sum(t, 0, i)
```

Si on calcule la WLP de la fin de la boucle vers le début, avec cet invariant, on obtient :

```
result + t[i] == partial_sum(t, 0, i + 1)
```

Ainsi, on voit que Frama-C aura besoin de vérifier cette propriété, mais la partie droite contient notre expression logique récursive, donc Frama-C aura du mal à la prouver sans axiome. Ainsi, un axiome intéressant ici, serait celui qui exprime exactement cette propriété (c'est `SumRec` en l'occurrence).

Appliquer les axiomes au fichier `TD3/src/ex4.c`.

a. En pratique, les gens sérieux (= pas nous dans ce cours) n'écrivent pas des axiomes gratuitement. Ils les prouvent avec des outils adaptés, par exemple en Coq.

Exercice 5

Compléter le fichier `TD3/src/ex5.c`.