

Conception Formelle

2021-2022

TP3: Boucles

Vincent Penelle

Points abordés

- Prouver des fonctions contenant des boucles.
 - Fonctions manipulant des tableaux.
 - Entiers non-signés.
-

Récupérer l'archive suivante¹ et décompressez-la.

Exercice 1: Ajoutons des boucles

1. Ouvrez dans Frama-C le fichier `identity.c`, et tentez de prouver les contrats avec WP. Ces contrats, bien qu'ayant l'air évidents, ne peuvent être prouvés. Cela est dû au fait que le nombre d'itérations d'une boucle est a priori inconnu (et on ne peut même pas décider, en général, si elle termine).

Point technique:

WP ne peut pas dérouler des boucles tout seul. Pour être capable de prouver des propriétés vérifiées par les boucles, on a besoin de la notion d'invariant de boucle. Un invariant de boucle est une propriété préservée par une application de la boucle, c'est-à-dire que si la propriété est vraie au début de la boucle, elle sera vraie à la sortie de la boucle. Il est possible de mettre plusieurs invariants de boucle (qui seront alors considérés comme devant tous être vrais (cf cours). La différence est que les preuves seront faites séparément et cela peut donc simplifier la tâche aux solveurs).

Exemple: `/*@ loop invariant 0 <= i < n;*/`

¹<https://www.labri.fr/perso/vpenelle/Enseignement/ConceptionFormelle/tp3.tar.gz>

2. Ajouter un invariant de boucle assurant que `result` reste plus petit que `n` dans la fonction `identity_while`. Voyez-vous pourquoi la fonction n'est toujours pas prouvée ? Vous pouvez essayer de placer un `assert` avec la post-condition à la fin de la fonction pour constater le problème.

 **Point technique:**

De même que pour les fonctions, Frama-C suppose qu'une boucle peut faire n'importe quel effet de bord sur la mémoire. Tout comme les fonctions, il est donc possible de déclarer (et prouver) des clauses `assigns` pour les boucles.

Exemple: `/*@ loop assigns i,n,max;*/`

3. Ajoutez le `loop assigns` pertinent dans `identity_while`. Frama-C devrait maintenant réussir à prouver le contrat de fonction.
4. Ajoutez maintenant un contrat de boucle pour la fonction `identity_for`. Attention, votre contrat de boucle devra aussi parler de `i` pour pouvoir être prouvé. De plus, prêtez attention qu'un invariant de boucle doit être vrai à l'entrée et à la sortie de la boucle, même après la dernière itération (lorsque la condition de la boucle est fausse).

Exercice 2: Comportements et boucles

 **Point technique:**

Lorsque les fonctions où se trouvent une boucles disposent d'un contrat avec des comportements, il est possible d'utiliser ces comportements dans les contrats de boucles pour en simplifier l'expression. De même que pour les contrats de fonctions, il est toujours possibles de les exprimer sans comportement. Il faut toujours écrire les clauses globales avant celles concernant des comportements.

Exemple: `/*@ for id: loop invariant i <= n; loop invariant j <= i; */`, où `id` est le nom d'un comportement.

1. Donner un contrat de boucle pour la fonction `behavioral_loop`, faisant intervenir les comportements définis dans son contrat.

Note : il est tout à fait possible d'obtenir un contrat de boucle valide en utilisant uniquement des implications. C'est d'ailleurs peut-être mieux : dans certaines fonctions, les solveurs n'arrivent pas à démontrer les versions avec `behavior` dans les boucles, mais y arrivent avec des implications. Ça doit venir de l'implémentation de tout ça, mais je ne suis pas certain du problème exact.

Exercice 3: Pour tout et il existe

 **Point technique:**

Dans certains contrats de fonction ou de boucles, on peut avoir besoin de parler d'un grand nombre de valeurs, ou de l'existence d'une valeur qu'on ne connaît pas exactement (par exemple, pour des propriétés parlant de tableaux, mais pas seulement). En ACSL, cela peut être exprimé à l'aide des constructions `\forall` et `\exists`. La syntaxe est `\forall type name; expr;`, où `type` est un type C ou mathématique (integer ou real, par exemple), `name` le nom de la variable que l'on quantifie et `expr` une expression ACSL (faisant intervenir `name`, sinon ce n'est pas très utile).

Exemple: `\forall int i; (0 <= i < n ==> \exists int j; 0 <= j < i && tab[j] < tab[i]);`

1. Donnez un contrat de fonction pour `set_to_zero` et validez-le. Pour réussir à le prouver, vous aurez besoin d'un contrat de boucle, avec un `loop assigns`.

 **Point technique:**

Attention, vous pourriez être tenté d'écrire un `loop assigns` de la forme `loop assigns tab[i]`. Cela n'est pas possible. En effet, le `loop assigns` dit que «pour toute valeur en mémoire non listée, elle est égale à la valeur qu'elle avait avant l'entrée dans la boucle avant et après le tour courant» (dit autrement, c'est un invariant de boucle un peu particulier). Or, si au tour `i`, on considère `tab[i]`, tous les `tab[j]` pour `j < i` ont été modifiés et ne sont plus égaux à la valeur qu'ils avaient avant le premier tour de boucle. Une autre manière de le voir est que ce contrat doit être utilisable par WP en abstrayant le contenu de la boucle (en regardant juste son contrat). Et la seule information pertinente dans ce cas seront quelles valeurs ont été modifiées à un tour de boucles quelconque, pas à quel tour de boucle elles l'ont été.

Exercice 4: Boucles et tableaux: spécification

 **Point technique:**

Pour la deuxième fonction vous pourriez (pour ne pas dire allez) avoir besoin de parler de la valeur d'une case mémoire lors du contexte d'appel de la fonction. Pour cela, vous ne pouvez pas utiliser `\old(x)`, mais vous pouvez utiliser `\at(x,Pre)` qui a le même effet.

1. Grâce à tout ce que vous avez vu jusqu'ici, déterminer et prouvez des contrats de fonctions et de boucles pour les deux fonctions présentes dans `exo4.c`.

Exercice 5: Attention aux entiers non-signés Puisqu'on manipule des tableaux, on pourrait vouloir considérer que la taille d'un tableau est un entier non-signé.

1. Lancez WP sur `signed-overflow.c` et observez que WP ne parvient pas à prouver les spécifications de `add` et `sub` (pour rappel, avec des `int`, il y arrivait, malgré la possibilité d'overflow).
2. Relancez l'analyse avec RTE activé. Rien ne change...



Point technique:

En C, sur les entiers non-signés, le comportement d'un overflow est bien défini (contrairement à l'overflow sur les entiers signés). Aussi, par défaut, RTE ne prévient pas des possibles overflow. WP en tient cependant compte, comme vous avez pu le constater. Il est possible de forcer RTE à générer des assertions pour les overflows d'entiers non-signés avec l'option `-warn-unsigned-overflow`.

3. Relancez WP avec RTE et l'option `-warn-unsigned-overflow`. Constatez la présence possible d'overflow.
4. Modifiez les contrats de fonctions pour supposer qu'il n'y a pas d'overflow, et constatez que WP parvient maintenant à prouver les contrat de fonction, même si on n'inclue pas les assertions de `-warn-unsigned-overflow`.

Exercice 6: Memcpy sûr Le but de cet exercice est d'utiliser tout ce que nous avons vus jusqu'ici pour concevoir une spécification formelle de la fonction `memcpy`, puis un code sûr l'implémentant.

1. Commencez par donner une spécification formelle de `memcpy` dans `memcpy.h` correspondant à la spécification informelle qui vous est donnée. Je vous conseille d'avoir au moins 2 comportements pour cette spécification. Vous veillerez à inclure une clause `assigns`. Pour vous aider, `memcpy.c` dispose d'un `main` contenant divers appels à `memcpy` suivis d'assertions qui doivent être vraie après l'appel de la fonction. Votre spécification devra permettre de valider ces assertions. Vous pouvez par exemple commencer par écrire la spécification et lancer `frama-c` avec cette spécification et sans implémentation de `memcpy` et vérifier si les appels dans `main` vérifient tous vos préconditions. Si ce n'est pas le cas, vos préconditions ne respectent pas ce qui est décrit dans la spécification informelle.

2. Codez maintenant la fonction `memcpy` de manière à ce qu'elle vérifie votre spécification. Attention, elle manipule des entiers non-signés, pensez à utiliser l'option `-warn-unsigned-overflow` pour être sûr de ne pas faire d'overflow involontaires.