Conception Formelle

2018-2019

TP4: Prédicats et fonctions (et lemmes, mais moins).

Vincent Penelle

Points abordés

- Abstraire une partie de la spécification à l'aide de fonction et de prédicats.
- Extraire certaines parties de preuves à l'aide de lemmes.
- Il ne faut pas non plus faire n'importe quoi.

Récupérer l'archive suivante[1] et décompressez-la.

Dans ce TP, dans un premier temps, vous ne préoccuperez pas des gardes RTE (ce n'est pas ce sur quoi je souhaite insister pour l'instant, et je n'ai pas pris le temps de faire en sorte que cela n'arrive jamais). Si vous terminez tout le sujet, vous pourrez revenir sur les exercices 1 et 2 et faire en sorte qu'aucune erreur de runtime ne soit possible si on respecte la spécification (soit en ajoutant des requires, soit en modifiant le code, lorsque cela est possible).

Exercice 1: Fonctions logiques

Point technique:

Il est possible de définir en ACSL des fonctions mathématiques. On peut le faire soit par une expression directe, soit via un ensemble d'*axiomes* permettant de la définir récursivement. Cette dernière version sera faite via un bloc dit "axiomatic".

Voir codes exemples de l'exercice courant.

L'intérêt de faire cela est double: il permet d'abord de n'écrire qu'une seule fois la définition de la fonction et d'utiliser ensuite le nom de la fonction (rendant ainsi la spécification plus lisible), mais aussi de définir des fonctions difficilement exprimables autrement (par ex, Fibonacci).

Bien évidemment, dans le dernier cas, il faudra veiller à définir des axiomes non-contradictoires (s'ils sont contradictoires, vous prouverez des trucs faux, ce qui est gênant). Par contre, les fonctions n'ont pas à être totalement définies, auquel cas on considère que la valeur renvoyée sur un cas non-défini est indéterminée. Veillez donc bien à ce vous n'appeliez de telles fonctions que sur des valeurs bien définie (faute de ne rien pouvoir prouver).

[1] `https://www.labri.fr/perso/vpenelle/Enseignement/ConceptionFormelle18/tp4.tar.gz`

Le fichier `exo1.c` contient des codes dont les contrats de fonctions vous sont fournis. Il vous «reste» à compléter les invariants (et variants) de boucle et les définitions des fonctions utilisées. Attention, sumOfInt contient un assert qui ne sera pas démontré sans ajouter de contrat de fonction, par contre, il aidera à démontrer ledit contrat de fonction. Ne paniquez donc pas si cet assert reste non-prouvé pour le moment.

1. Observez les définitions de `nextInt` et `fibonacci`, et observez que les fonctions correspondantes sont prouvées.

2. Écrivez des définitions pour les fonctions `sumOfInt` et sumFirstLast, et prouvez les contrats des fonctions `sumInt` et `sumFL`.

3. La spécification de `sumFLBis` n'est pas prouvée. C'est normal, puisque le tableau a été modifié après qu'on ait calculé la valeur. On ne demande pas de corriger le code.

Point technique:

Il est possible de demander l'évaluation d'une fonction en un point particulier du programme, en lui fournissant un label L. La fonction sera alors calculée en ce label. Les règles s'appliquant aux labels possibles sont les même que pour \at (i.e., pas de labels à l'intérieur de blocs, pas de labels après l'appel).

Syntaxe: `\result == maFonction{Pre}(t,i,n);` évaluera maFonction avant l'appel de la fonction.

C'est bien évidemment très utile pour les pointeurs ou tableaux, puisque remplacer `t` par `\at(t,Pre)` dans le contrat de `sumFLBis` ne changera pas que les valeurs pointées seront celles après l'appel de la fonction.

4. Corrigez la spécification de `sumFLBis`.

Point technique:

Vous noterez la présence d'un terme `reads t[i .. n-1]` dans les définitions des prédicats `sumOfTab` et `countTab`. Cela indique les éléments dont a besoin la fonction pour être calculée. Cela permet de rendre la spécification plus lisible.

Cela a en fait une conséquence pratique (qui semble dépendre de la version d'alt-ergo): si vous ne le spécifiez pas, frama-c aura beaucoup plus de mal à démontrer quoique ce soit faisant intervenir la fonction. Notamment, si vous l'enlever de ces deux fonctions, vous ne réussirez plus à prouver les contrats de fonction les utilisant. Ce problème n'apparaît pas avec alt-ergo 2.3.

5. Définissez les fonctions `sumOfTab` et `countTab` en définissant les axiomes décrits informellement, et vérifiez qu'elles permettent de vérifier les contrats de `sumOfTab` et `countTab` (notez au passage qu'une fonction C et une fonction logique peuvent avoir le même nom).

6. Corrigez la spécification de `sumOfTabInTab` et prouvez-la (cf points 3-4).

Exercice 2: Prédicats

Point technique:

Un prédicat peut simplement être vu comme une fonction à valeur de sortie booléenne. Ils se définissent de manière similaire aux fonctions (simplement, leur type de sortie n'est pas spécifié, et il sont définis comme une formule logique, et non comme une valeur typée, comme les fonctions).

Tout comme les fonctions, ils ne sont pas nécessairement définis totalement auquel cas dans les cas non-définis la valeur est indéterminée. Vous pourrez observer que si on enlève le dernier axiome concernant fiboTab, le contrat de fonction n'est pas prouvé, puisque frama-c n'arrive pas à prouver que le prédicat est faux dans les cas non-spécifiés par les deux premiers axiomes.

1. Observez les définitions de `isPositive` et `fiboTab`, et observez que les fonctions correspondantes sont prouvées.

2. Écrivez des définitions pour les fonctions `isBiggerThanTab` et tabNextInt, et prouvez les contrats des fonctions `ceilTab`. et `addOneInTab`.

3. La spécification de `addOneTab` n'est pas prouvée. C'est normal, puisque le prédicat reçoit deux fois le même tableau. Comme pour les fonctions, l'appeler sur `\at(t,Pre)` et `\at(t,Pre)` ne changerait rien, puisque ces deux valeurs pointent au même endroit, et que dans tous les cas, on les évaluera à Post. On ne demande pas de corriger le code.

   

   Point technique:

   Tout comme pour les fonctions, il est possible d'appeler un prédicat sur un label du programme. On peut même définir les prédicats (et les fonctions aussi) pour qu'ils lisent des valeurs à différents labels, mais il faut pour cela l'indiquer dans la définition du prédicat.

   Exemple: `predicate toto{L,M}(integer i) = \at(i,L) == \at(i,M)` sera vrai si et seulement si `i` vaut la même valeur aux labels L et M.

   Attention, si vous définissez plusieurs labels dans un prédicat (ou fonction), toutes les variables devront être dans un \at (sinon c'est ambigu).

   Comme pour les fonctions, pour les prédicats définis via des axiomes, on peut indiquer à la déclaration quelles valeurs sont lues par le prédicat grâce au mot clé reads.

   Exemple: `predicate tata{L,M}(int *t, integer n) reads \at(t[0],L), \at(t[n-1],M);` utilisera uniquement les valeurs de t[0] en position L et de t[n-1] en position M.

   Vous ne devriez pas réussir à démontrer la fonction `threeSwap` sans reads dans `permutation` (et là, même avec alt-ergo 2.3).

4. Corrigez la spécification de `addOneTab`, en définissant les prédicats `tabNextIntFixed` et `unchangedTab` (le premier est à prouver pour le contrat de fonction, le second sera utile pour un invariant de boucle).

5. Définissez les prédicats `swapInArray` et `permutation` de manière à prouver les fonction `swapArray` et `threeSwap`. Des descriptions informelles des axiomes nécessaires sont donnés pour vous aider.

Exercice 3: Lemmes

Point technique:

Les lemmes sont des formules logiques que l'on peut démontrer comme étant tout le temps vraies (en fait c'est comme un théorème, mais avec un autre nom). Pour frama-c, l'intérêt d'écrire des lemmes peut être de permettre aux solveurs de démontrer plus facilement un contrat de fonction (ou d'y arriver tout court). Il est bien évidemment nécessaire de les démontrer par ailleurs (et on peut (doit) demander aux solveurs de les démontrer).

Syntaxe: `/*@ lemma nameOfLemma: \forall integer i; i < i+1;*/`

À noter qu'il est également possible d'indexer des lemmes par des labels abstraits (cf TP3 exo 4 pour la notion de labels) permettant de parler d'une même valeur à plusieurs points du programme. Ça peut avoir l'air inutile à première vue, mais c'est plus flagrant si on y met des prédicats ou des fonctions.

Exemple:
```
/*@ lemma toto{L1,L2,L3}: \forall int *t;
\at(t[0],L2) == (\at(t[0],L1) + \at(t[0],L3)/2)
==> \at(t[0],L1) <= \at(t[0],L2) <= \at(t[0],L3);
```

1. Recopiez les lemmes de l'exemple dans un fichier vierge `exo3.c`, et observez que le premier est démontré, mais pas le deuxième (c'est normal, il est faux).
2. Corrigez le deuxième lemme pour qu'il soit démontré par frama-c (plusieurs solutions sont possibles, vous pouvez renforcez la condition en imposant l'ordre entre L1 et L2, ou affaiblir le conséquent en disant que soit l'ordre présenté est vrai, soit son inverse.
3. Écrivez et démontrez un lemme disant que pour tout entier, son carré est positif.
4. Écrivez et démontrez un lemme disant que pour tout entier, son carré est nul si et seulement si il est nul.
5. Écrivez et démontrez un lemme disant que tout int est compris entre INT_MIN et INT_MAX (pensez à inclure limits.h).
6. Écrivez et démontrez un lemme disant que pour tout couple de pointeurs p et q, si à un label L1 *p <= *q, et qu'à un label L2, *p reste inchangé mais que *q a augmenté, alors l'inégalité reste vraie au label L2.
7. (si vous arrivez là en un quart d'heure ou moins) Écrivez et démontrez un lemme disant que pour tout tableau d'entier t et pour tout entier n, si n est strictement plus grand que 0, alors il existe un indice max entre 0 et n-1 tel que t[max] est plus grand que tous les t[i] pour i entre 0 et n-1.

Il est bien évidemment possible de définir des lemmes faisant appel à des fonctions ou des prédicats définis préalablement (comme dans les exos précédents). C'est même l'un des principaux intérêts. Mais comme j'ai écrit cet exercice avant les précédents avant de

me rendre compte que c'était mieux dans cet ordre là et que je n'ai plus trop le temps d'en rajouter, je ne mets pas d'exemples avec prédicats fonctions pour l'instant. Mais n'hésitez pas à essayer.

Exercice 4: Lemmes: ne pas faire n'importe quoi Le fichier `exo4.c` contient un code ne respectant pas sa spécification (sans blagues). Ajoutez à ce fichier (avant la fonction) un lemme disant que pour tout entier, s'il est égal à zéro, alors il est différent de zéro, et observez que frama-c réussi maintenant à prouver que cette fonction est correcte.

La raison en est évidemment que le lemme étant contradictoire, il en a déduit faux.

Toutes les erreurs ne seront évidemment pas si grossières. Aussi, si vous arrivez à prouver un résultat avec un lemme non prouvé, ne considérez pas que votre fonction est prouvée (ou seulement si vous êtes capables de prouver ces lemmes par ailleurs)!

Bien évidemment, si vous définissez une axiomatique contradictoire (par exemple en définissant une fonction ou un prédicat), vous aurez le même genre de blague, mais cette fois-ci, frama-c ne vous préviendra pas. Faites donc bien attention, et en cas de doute, tentez de prouver un lemme faux (ou directement faux, mais ça ne fonctionne pas toujours), pour vérifier que vos axiomes ne sont pas contradictoires.