Conception Formelle

2022-2023

TP5: Fonctions logiques et prédicats : formalisation plus poussée.

Vincent Penelle

Points abordés

- Abstraire une partie de la spécification à l'aide de fonction et de prédicats.
- Comment utiliser différentes positions du programmes dans les fonctions et les prédicats.
- Définir des fonctions et prédicats via des axiomatiques.

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

Exercice 1: Fonctions logiques : mise en jambes

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" (on verra cela dans un exercice ultérieur).

Voir code exemple de l'exercice courant.

Le premier intérêt de faire cela est, comme pour les prédicats, 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).

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. Un des deux ensures de `doublePointer` est volontairement fausse (cf. deuxième point technique), donc il est normal qu'il ne soit pas prouvé, et il ne vous est pas demandé de le corriger.

1. Observez la définitions de `nextInt`, et observez que la fonction correspondante est prouvée.
2. Écrivez des définitions pour les fonctions `sumOfInt` et `sumFirstLast`, et prouvez les contrats des fonctions `sumInt` et `sumFL`. N'oubliez pas d'enlever l'espace devant le @ des spécifications des fonctions que vous voulez prouver après avoir écrit vos fonctions logiques (je les ai mis pour que frama-c accepte d'ouvrir le fichier, puisque les fonctions logiques ne sont pas définies).

[1] `https://vpenelle.pages.emi.u-bordeaux.fr/conception-formelle/tp5.tar.gz`

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, mais la spécification : il suffit de dire que la fonction `sumFirstLast` est évaluée au début du programme avec \old.

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 considérant que les valeurs pointées sont évaluées au label L. 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).

Exemple: `\result == sumFirstLast{Pre}(t,n);` évaluera sumFirstLast en regardant les valeurs pointées par t avant l'appel de la fonction C, mais en évaluant n après l'appel de la fonction C (la fonction C où cette spécification est mise).

Cette syntaxe n'est pas équivalente à `\result == \old(sumFirstLast(t,n);)` : le label ne concerne en effet que les valeurs pointées dans le corps du prédicat, pas dans ses arguments, contrairement au `\old` qui affectera tous les arguments. En l'espèce, si on remplace maintenant n par une valeur pointée *i, avec un old, *i serait évaluée avant l'appel de la fonction, alors qu'avec {Pre}, il le serait à la fin. Voir la spécification de `doublePointer` pour avoir un aperçu du phénomène (il est normal que seule l'une des deux post-condition soit vraie).

`\old(sumFirstLast(t,n))` est équivalent à `sumFirstLast{Pre}(\old(t),\old(n))`.

4. Remplacez le \old par la syntaxe décrite ci-dessus, puisqu'en réalité c'est le comportement souhaité (certes, pour ce cas précis, puisque n est un argument de la fonction, ça ne change rien). Vous remarquerez que `doublePointerBis` est une meilleure spécification (mais le but de ces mauvais exemples est de vous faire jouer avec).

Exercice 2: Labels multiples et prédicats

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}(int *i) = \at(*i,L) == \at(*i,M)` sera vrai si et seulement si `i` pointe vers la même valeur aux labels L et M.

Attention, si vous définissez plusieurs labels dans un prédicat (ou fonction), toutes les valeurs pointées devront être dans un \at (sinon c'est ambigu), les arguments entiers étant déterminés à l'appel de la fonction, ce n'est pas la peine de les mettre dans des \at.

1. Après avoir regardé les exemples de prédicats, définissez les deux prédicats unchangedTab et tabNextIntFixed et ajoutez les bons invariants de boucles pour prouver la spécification de la fonction addOneTab.

Exercice 3: Définitions axiomatiques

Point technique:

Certaines fonctions ou prédicats ne sont pas aisément définissables par une expression logique ou arithmétique, mais sont plus faciles à définir de manière récursive. Par exemple, la somme des éléments d'un tableau ne peut pas être écrite via une seule expression, puisque cela dépend du nombre d'élements dans le tableau, mais est assez facile à définir si on définit la somme des éléments d'un tableau vide (0) et la somme des éléments d'un tableau de taille n est la somme des éléments du tableau de taille n-1 qui ne contient pas la dernière case, plus cette dernière case.

En Frama-C, on peut définir des fonctions et prédicats de cette manière grâce à des axiomatiques (et même autrement que récursivement) qui se définissent globalement avec les éléments suivants :

- D'abord un nom (pour les identifier).
- Ensuite une ligne déclarant une fonction ou un prédicat exactement de la même manière qu'on les définirait normalement, mais sans expression logique les définissant. Cette déclaration peut éventuellement être suivie d'un terme `reads` qui indique quelles sont les valeurs mémoire utilisée dans la définition du prédicat ou de la fonction. L'intérêt d'un tel terme est d'aider à la preuve des assertions faisant intervenir ce prédicat ou cette fonction en simplifiant un peu les hypothèses inutiles. C'est donc parfois important pour les preuves, mais pas pour la définition.
- Enfin des axiomes définis avec le mot clé axiom, suivi d'un nom et d'une expression logique. Dans l'idéal, cette expression devrait permettre de définir partiellement la fonction ou le prédicat (on peut mettre n'importe quoi comme axiome, mais dans une axiomatique définissant un prédicat, il est pertinent que les axiomes en parlent). On peut (et très souvent doit) utiliser plusieurs axiomes pour définir le comportement d'un prédicat ou d'une fonction. Ces axiomes seront considéré comme vrai par Frama-C.

Il faut faire très attention avec les axiomatiques : comme Frama-C considère que ce qui y est écrit est vrai, il faut veiller à ne pas écrire d'axiomes contradictoires, et bien veiller à ce qu'ils ne permettent que de définir ce que vous cherchez à définir. En effet, si des axiomes sont contradictoires, alors Frama-C réussira à prouver des trucs faux, et donc n'importe quoi !

On donne des exemples dans le code de cet exercice.

Note, dans cet exercice, ne vous préoccupez pas des gardes RTE, elles ne sont pas évidentes à prouver, et ce n'est pas ce qui est important ici.

1. Regardez et comprenez les définitions axiomatiques Fibo et FiboTab.
2. Définissez les axiomatiques SumTab et CountTab (des indications vous sont données pour les axiomes), et déterminez et prouvez les bons invariants pour les fonctions faisant intervenir ces fonctions.

3. Définissez un prédicat swapInArray. Je vous donne des indications, et vous encourage à le découper en sous prédicats. Ce prédicat peut être défini sans axiomatique. Vous pourriez avoir besoin du prédicat unchangedTab défini à l'exercice 1. Utilisez-le pour prouver la spécification de la fonction swapInArray.

4. Définissez maintenant l'axiomatique Permutation (encore une fois, vous avez des indications). Observez que vous arrivez à prouver la spécification de threeSwap. Vous pourrez voir que vous n'arrivez pas à prouver `ensures permutation{Pre,Post}(t,t,i,n)` dans la fonction swapInArray. C'est parce que les prédicats définis de ce genre ne sont pas aisé à démontrer si on n'a pas des axiomes sur mesure, et en particulier dans une fonction qui modifie un tableau à plusieurs endroit du code, il est difficile de prouver un tel prédicat. Aussi, pour les preuves, il sera plus aisé de découper le code et d'utiliser swapInArray à chaque fois que c'est possible pour faciliter les preuves. D'une manière générale, il est plus facile de prouver de petites fonctions.

Exercice 4: Axiomatiques : de l'art de choisir la bonne

Attention : Pour cet exercice, Alt-ergo n'arrive pas à démontrer les propriétés demandées pour l'axiomatique BetterCountTab. Par contre Z3 y arrive, lui.

Pour définir une fonction, on a souvent plusieurs choix possible pour les définir, et toutes ces définitions ne permettent pas de démontrer toutes les propriétés qu'on voudrait. Cela n'est pas dû forcément à une mauvaise définition, mais plutôt à la difficulté de démontrer automatiquement des propriétés, notamment avec des axiomes un peu trop spécifiques. Le but de cet exercice est de remarquer ce fait.

- Recopiez vos spécifications pourt countTab et l'axiomatique CountTab. Déterminez ensuite les invariants de boucle pour countTabRev, et observez que Frama-C n'arrive pas à les démontrer. C'est dû à ce qu'on décrivait plus haut (si countTab est correct, votre axiomatique l'est très probablement).

- Ajoutez deux axiomes permettant de décrire le comportement de countTab quand on fait la somme par la gauche (similaire à votre version) et quand on fait la somme des résultats de deux tableaux. Vos fonctions devraient maintenant être démontrées.

- Désactivez votre axiomatique CountTab, et écrivez une axiomatique BetterCountTab en utilisant les indications données dans le fichier. L'idée est de définir countTab pour les tableaux de taille 1, et de définir que pour un tableau plus grand, on peut obtenir le résultat en coupant le tableau en 2 (n'importe où). Observez que countTab et countTabRev sont maintenant démontrées (avec Z3).

- Vous pouvez aussi définir les invariants de boucles pour countTabTer et remarquer que la nouvelle axiomatique permet de le démontrer.

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 plus haut).

- Pour avoir utilisé une fois des lemmes et vérifié que votre nouvelle axiomatique est bien pertinente, vous pouvez mettre les axiomes de CountTab comme des lemmes, et remarquer qu'ils sont bien démontrés par Z3.