Conception Formelle

2018-2019

TP3: Terminaison, correction partielle et totale, et récursion.

Vincent Penelle

Points abordés

- Terminaison de boucles.
- Correction partielle et correction totale.
- Fonctions récursives.

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

À partir de maintenant, j'oublierai très régulièrement de vous demander de penser à prouver en plus du reste que la fonction ne provoque pas d'erreur de runtime (i.e., de prouver que les assertions générées par RTE sont prouvées). Faites-le quand même.

Exercice 1: Une boucle infinie triviale.

1. Ouvrez le fichier `non-terminate.c` et prouvez le contrat de fonction avec WP. Observez que toutes les assertions sont prouvées.
2. Ajoutez `/*@assert \false;*/` après la boucle, et observez que cette assertion est également prouvée.

Point technique:

Il est facile d'observer (du moins j'espère que vous l'avez vu), que la boucle de la fonction `f` est une boucle infinie et qu'on ne sortira jamais de la fonction. Dans ce cas, WP est capable de détecter que la partie de code située après est inatteignable et considèrera que toute assertion y étant mise est vraie. Même Faux!

Exercice 2: Terminaison de boucle.

La fonction située dans `terminating .c` termine trivialement (même si ce n'est pas un for). Le but de cet exercice est de vous montrer comment écrire une telle assertion.

1. Pour vous dérouiller depuis la semaine dernière, commencez par ajouter les invariants de boucles et assigns nécessaire pour que le contrat de fonction soit prouvé.

[1] `https://www.labri.fr/perso/vpenelle/Enseignement/ConceptionFormelle18/tp3.tar.gz`

Point technique:

Un *variant de boucle* est une valeur entière décroissant *strictement* à chaque tour de boucle, tout en restant positif (sauf éventuellement à la fin du dernier tour de boucle). La présence d'un tel variant assure la terminaison de la boucle, puisque que pour tout entier, il existe un nombre *fini* d'entiers plus petits.

Syntaxe: `/*@ loop variant i;*/` ou `/*@ loop variant 2*n - i + 4;*/`.

2. Ajoutez un variant de boucle à votre fonction pour démontrer qu'elle termine.

Exercice 3: Corrections partielles et totales. La correction partielle d'une boucle/fonction consiste à prouver que le contrat est satisfait *si la fonction termine*, mais n'assure pas la terminaison.

La correction totale consiste à démontrer que de plus la boucle/fonction termine sur toutes ses entrée (et donc que le contrat de fonction est toujours satisfait). Pour les boucles, ce dernier point peut être prouvé à l'aide d'un variant.

Pour des fonctions non-récursives, on considèrera que la terminaison est démontrée si la terminaison de toutes ses boucles l'est (WP ne sait malheureusement pas le démontrer automatiquement).

1. Prouvez avec WP la fonction contenue dans le fichier `max_tab.c`. Remarquez que tout est prouvé sans problème. Observez que si on ajoute des demandes absurdes, elles ne sont PAS prouvées, contrairement au cas de l'exercice 1. Cela est dû au fait que la non-terminaison de la boucle n'est pas triviale.

2. Tentez d'ajouter un variant de boucle pour prouver la terminaison de cette boucle. Est-ce normal (c'est-à-dire, cette boucle termine-t'elle) ?

3. Corriger le code et prouvez la terminaison de votre boucle. Bravo, vous venez de démontrer la correction totale de votre fonction `max_tab` corrigée.

4. Ouvrez maintenant le fichier `mystery.c` et prouvez sa correction partielle. La demande d'un entier à un joueur est modélisé par la fonction askPlayerNumber dont le code n'est pas donné, mais la spécification est fournie. Indice, vous n'avez pas besoin d'invariant ici.

   Pourriez-vous prouver la correction totale de cette fonction ? Pourquoi ?

Exercice 4: Doublement non-déterministe On considère la fonction contenue dans le fichier `non-det.c` (oui, je sais, les fonctions de ce TP sont bidons, mais les exemples intéressants avec des boucles vont être durs à prouver au début). On souhaite en prouver la correction totale, et, le cas échéant la corriger. Comme dans l'exercice précédent, on simule un générateur aléatoire de nombre par une fonction dont on a uniquement donné la spécification.

1. Prouvez la correction partielle de cette fonction. Pour cela, vous aurez besoin de la valeur de `a` avant le début de la boucle (cf point ci-dessous).

Point technique:

Il est possible de mettre des contraintes sur des valeurs de variables située ailleurs qu'à la ligne courante. C'est implicitement ce qui est fait dans les contrats de fonctions (c'est pour cela que des `\old` y apparaissent). Pour cela, on peut le spécifier avec l'instruction `\at(var,Label)`, où `var` est la variable considérée et `Label` une étiquette située à une ligne du code (pour rappel, une étiquette est de la forme `MonLabel:` au début d'une ligne). Deux conditions cependant: on ne peut référer qu'à une étiquette située plus haut dans le programme, et pas à une étiquette interne à un bloc.

En plus des étiquettes définies par l'utilisateur, il existe un certain nombre de mots-clés qui peuvent remplacer les étiquettes pour désigner des places prédéfinies du code relativement à la ligne de l'assertion:

- Here. Désigne la ligne de l'assertion. `\at(toto,Here)` est généralement équivalent à écrire `toto` (je n'ai pas trouvé de contre-exemple. Utile pour clarifier.
- Old. Uniquement accessible dans les clauses ensure et assigns d'un contrat. Désigne l'état d'une variable avant le contrat.
- Post. Similaire à Old, mais désigne l'état d'une variable après le contrat.
- Pre. Visible partout. Correspond à l'état d'une variable avant l'appel de la fonction (disponible uniquement sur les arguments et variables globales, donc).
- LoopEntry. Visible dans une boucle. Réfère à l'état d'une variable juste avant le début de la boucle où elle apparait (mais après l'initialisation d'un for).
- LoopCurrent. Visible dans une boucle. À peu près équivalent à Here (sauf dans un cas expérimental d'après la doc).
- Init. Visible partout. Réfère à l'état d'une variable globale avant l'appel à la fonction main.

2. Tentez de prouver la correction totale. Est-ce possible ?

3. Corrigez le code pour que la correction totale soit vraie, et prouvez-la.

Exercice 5: Correction totale générales

On a vu que les variants de boucles étaient nécessairement des entiers qui décroissaient à chaque tour de boucle.

En réalité, un variant pourrait être n'importe quel type de données pourvu que sa valeur décroisse strictement à chaque tour de boucle selon un *ordre bien fondé* (c'est-à-dire, que pour tout élément, il y a un nombre fini d'éléments plus petits (je mens un peu ici, mais c'est l'idée – si vous voulez la vraie définition, demandez-moi, au tableau, ça passe mieux)).

ACSL permet d'exprimer cela avec l'expression: `/*@ loop variant var for relation; */`, où `relation` est un ordre bien fondé sur le type de `var` (ce qui, bien évidemment, devrait être prouvé par ailleurs).

Génial!

Bon en fait, s'il est possible de l'écrire et que Frama-C le parse, WP n'est actuellement pas capable de traduire une assertion de cette forme en obligation de preuve.

- Toutefois, ouvrez le fichier `max_matrix.c` pour voir comment il est possible de l'écrire. Ne faites pas trop attention aux axiomatiques et prédicats: ce sont des manières de définir des fonctions logiques en Frama-C sur lesquels on reviendra plus tard. Si on a le temps et que le reste est clair.
- (Bonus) Cela dit, vous noterez que je n'ai mis que le variant de boucle ici. Donc, je n'ai pas prouvé la correction partielle de ma super fonction sur des matrices. Allez, au boulot !

Exercice 6: Fonctions récursives: correction partielle, mais pas totale
Frama-C est également capable de démontrer la correction de fonctions récursives.

1. Donnez des spécifications pour ces fonctions récursives et prouvez-les.
2. N'oubliez pas les gardes RTE (et pour cela, vous aurez peut-être besoin d'aider WP avec des assert).

Point technique:

Comme pour les boucles, il existe en ACSL une manière de spécifier qu'une fonction termine, et également qu'une valeur entière décroit strictement à chaque appel récursif d'une fonction donné (ce deuxième point uniquement pour les fonctions récursives).

`/*@ terminates cond;*/` indique que la fonction termine toujours lorsque la propriété `cond` est vérifiée.

`/*@ decreases expr;*/` indique qu'à chaque appel récursif de fonction, la valeur de l'expression `expr` décroit strictement (exactement comme pour un variant de boucle).

Je sens que vous le voyez venir : comme pour les variants généraux pour les boucles, si ces propriété sont exprimables, WP n'est à ce jour pas capable de les traduire en obligation de preuves. Vous pouvez toutefois annoter vos fonctions avec ces mots clés pour indiquer qu'elles devraient terminer.

Exercice 7: Produire du code sûr avec des boucles Spécifiez, implémentez et prouvez la correction totale et l'absence d'erreur de runtime des fonctions dont les prototypes sont dans le fichier `exo7.h`. Vous n'implémenterez pas la fonction randInt.

Exercice 8: Revenons un peu en arrière Au TP2, nous n'avons prouvé que la correction partielle des fonctions qu'on a manipulé. Prouvez leur correction totale maintenant.