Conception Formelle

2022-2023

TP4: 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. Cela est dû au fait que la condition de répétition de la boucle est FAUX. Dans ce cas, l'algorithme WP implique bien n'importe quelle formule en fin de boucle (puisque la formule à démontrer est $I \wedge \neg \text{test} \Rightarrow \varphi$). Si on ne met pas d'invariant (ce qui est le cas ici), on a $I = \top$, ce qui permet de comprendre pourquoi Frama-c n'a aucun problème à valider cette spécification.

3. Ajoutez `terminates \true;` au début du contrat de fonction de f, et observez que cette clause n'est pas prouvée. Cela veut dire que Frama-C n'arrive pas à démontrer que cette fonction termine (et pour cause).

[1] `https://vpenelle.pages.emi.u-bordeaux.fr/conception-formelle/tp4.tar.gz`

Point technique:

La clause `terminates phi;` exprime que la fonction termine sur tous les contextes d'appels dans lesquels la formule `phi` est vraie. En particulier, `terminates \true;` est vrai si la fonction termine dans tous les contextes d'appels. Il peut y avoir un intérêt à mettre autre chose que `\true` comme argument quand on veut bien spécifier que le comportement normal de la fonction n'est pas de terminer dans tous les cas (mais le plus souvent on voudra qu'elle termine tout le temps).

À noter que `terminates \false;` est toujours vrai (et ne veut pas dire que la fonction ne termine pas).

Le support de la preuve de terminaison par Frama-C date de la version 24.0 et est a priori encore expérimental.

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.

2. Ajoutez une clause `terminates \true`. Frama-C n'arrive pas à la démontrer, car il n'a pas assez d'informations pour cela.

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;*/`.

Si toutes les boucles d'un programme disposent d'un variant démontré, et que toutes les fonctions appelées terminent également, alors Frama-C pourra certifier que la fonction termine (en validant la clause terminates).

3. 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 (ce qu'on peut vérifier grâce à la clause terminates).

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.

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: Terminaison mais pas tout le temps Pour cet exercice, on va prendre l'exemple de la division euclidienne qu'on a vu en cours. Le but est de voir un truc pour montrer une terminaison dans un certain sous-domaine de définition sans avoir à spécifiquement interdire ce cas dans les requires. Pour l'exemple de la division euclidienne, il serait évidemment mieux de réduire le domaine de définition (puisque la fonction ne termine jamais), mais on pourrait avoir des cas où une fonction doit terminer dans un certain sous-domaine, mais pas forcément dans un autre (par exemple parce que la terminaison dépend d'une interaction avec l'environnement).

1. Lancez frama-c sur `euclidean.c`, et observez que le contrat qui y est donné est vérifié par frama-c (sauf la terminaison). Attention, pour une raison qui m'échappe, alt-ergo n'arrive pas à tout démontrer, mais Z3, oui.

2. Trouvez un variant de boucle qui permet de montrer la terminaison dans le cas où $b > 0$ (comme vu en cours). Frama-C n'arrivera pas à le démontrer puisque quand $b \leq 0$, ce variant n'en est pas un (il ne décroit pas strictement).

3. Relancez frama-c avec l'option `-wp-variant-with-terminates`. Frama-c devrait maintenant être en mesure de prouver la terminaison de votre fonction.

Point technique:

L'option `-wp-variant-with-terminates` change la génération des obligation de preuve des variants : si elle est présente, les variants ne seront prouvés que dans le cas où l'hypothèse de la clause `terminates` est satisfaite (contre dans tous les cas où les `requires` sont satisfaits en son absence). En particulier, cela veut dire que si on appelle cette option avec un `terminates \false;` tous les variants seront trivialement vrais, et avec un `terminates \true;`, cette option n'a pas d'intérêt.

Vous pouvez consulter le manuel de WP[a] (page 37) pour plus de détails.

[a] `https://frama-c.com/download/frama-c-wp-manual.pdf`

Exercice 5: 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). Vos invariants doivent parler de la valeur courante de `res`, la valeur courante de `a` et la valeur de `a` au début de la fonction (`\at(a,Pre)`).

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. `\at(toto,Old)` est équivalent à `\old(toto)`.
- 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. Attention, on ne demande PAS de modifier la spécification de `randInt` (imaginez que c'est une fonction d'une bibliothèque), mais son utilisation dans `nondet`.

Exercice 6: Tous les variants ne sont pas (si) simples On a vu que les variants de boucles étaient nécessairement des entiers qui décroissent à chaque tour de boucle. Souvent, il est aisé de les déduire immédiatement de la condition d'arrêt de la boucle, mais ce n'est pas toujours le cas. Des fois, il serait nécessaire d'utiliser un ordre sur un ensemble différent de $\mathbb{N}$, mais, malgré cela, il est souvent possible de définir une fonction de cet ensemble dans $\mathbb{N}$ qui, elle, va strictement décroître à chaque tour de boucle (vous vous souvenez le TD dénombrement en CoCa ? Ben voilà).

- Ouvrez le fichier `max_matrix.c` et déterminez, à partir de la manière dont est explorée la matrice, un variant de boucle. Attention, alt-ergo n'arrivera pas à le démontrer, mais Z3 lui y arrive.
- (Bonus) Je n'ai pas démontré la correction partielle de cette fonction, ni donné de post-condition. S'il vous reste du temps, faites-le.

Exercice 7: 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.

Point technique:

Comme pour les boucles, il existe en ACSL une manière de spécifier qu'une fonction récursive termine : pour cela, il faut donner une valeur entière décroit strictement à chaque appel récursif d'une fonction donné (ce qui est très similaire à un variant de boucle).

`/*@ 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 le traduire en obligation de preuves. Vous pouvez toutefois annoter vos fonctions avec ce mot clé pour indiquer qu'elles devraient terminer, d'autant qu'en présence de telles clauses, Frama-C acceptera de valider vos clauses `terminates` (en vous disant qu'il manque une preuve, mais là vous n'y pouvez rien).

3. Trouvez des clauses `decreases` qui permettront de valider les clauses `terminates \true` de chaque fonction.

Exercice 8: 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 `exo8.h`. Vous n'implémenterez pas la fonction randInt.

Exercice 9: Revenons un peu en arrière Au TP3, nous n'avons prouvé que la correction partielle des fonctions qu'on a manipulé. Prouvez leur correction totale maintenant.