TP 7 : Logique de Hoare, vérification de programmes

1 Logique de Hoare, correction partielle et correction totale

On rappelle les règles définissant le jugement $\vdash \{A\}c\{A'\}$, correspondant à la correction partielle des programmes Imp :

$$\frac{}{\vdash \{A\}\texttt{skip}\{A\}} \qquad \frac{}{\vdash \{A[a/X]\}X := a\{A\}} \qquad \frac{\vdash \{A\}c_1\{A'\} \qquad \vdash \{A'\}c_2\{A''\}}{\vdash \{A\}c_1; c_2\{A''\}}$$

$$\frac{\vdash \{A \wedge b\}c_1\{A'\} \qquad \vdash \{A \wedge \neg b\}c_2\{A'\}}{\vdash \{A\}\texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\{A'\}} \qquad \frac{\vdash \{A \wedge b\}c\{A\}}{\vdash \{A\}\texttt{while } b \texttt{ do } c\ \{A \wedge \neg b\}}$$

$$\frac{\vdash \{A_0\}c\{A'_0\} \quad \models A \Rightarrow A_0 \quad \models A'_0 \Rightarrow A'}{\vdash \{A\}c\{A'\}}$$

La dernière règle, appelée *règle de conséquence*, s'appuie sur la démonstration « en mathématiques » d'implications entre les assertions concernées.

Correction des règles de la logique de Hoare. On montrera en cours la **correction de la logique de Hoare**, qui s'énonce

$$\vdash \{A\}c\{A'\} \quad \Rightarrow \quad \models \{A\}c\{A'\} \ ,$$

autrement dit, que tout triplet dérivable est valide.

Corrections partielle et totale. La validité fait ici référence à la correction partielle pour le triplet $\{A\}c\{A'\}$, qui s'énonce ainsi :

> *Pour tout état mémoire σ, si σ satisfait A et si l'exécution de c à partir de σ termine, en donnant un état σ', alors σ' satisfait A.*

La *correction totale* pour le triplet $\{A\}c\{A'\}$ s'énonce en revanche

> *Pour tout état mémoire σ, si σ satisfait A, alors l'exécution de c à partir de σ termine, en donnant un σ' satisfaisant A dans V.*

En d'autres termes, on prouve à la fois une propriété mettant en relation la précondition et la postcondition, et la terminaison du programme. On note $[A]c[A']$ pour le triplet de Hoare interprété avec la correction totale.

La correction totale des programmes s'établit comme la correction partielle, excepté pour la commande `while`, qui est la seule qui peut introduire des divergences. On adopte pour ce cas la règle suivante

$$\frac{[A \wedge b \wedge v = Z \wedge Z \geq 0]\, c\, [A \wedge v < Z]}{[A]\texttt{while } b \texttt{ do } c\ [A \wedge \neg b]} Z \notin \mathrm{Vars}(\texttt{while } b \texttt{ do } c\ )$$

A est ici l'*invariant* de boucle, et v est le *variant*. Z est une *"spurious variable"*, autrement dit une variable qui ne sert qu'au raisonnement (puisque le programme ne la manipule pas).

Question 1. Écrivez une version annotée du programme Imp suivant :

```
r := 0;
k := 0;
while k <= n do
  r := r + k;
  k := k + 1
```

Soyez convaincu(e) que vous savez décrire comment on construit la dérivation correspondante à l'aide des règles de la logique de Hoare.

Question 2. On se pose la question de l'automatisation de la construction de dérivations en logique de Hoare.

1. Soit c une commande Imp ne comportant ni test, ni boucle, et soient A et A' deux assertions.
   Comment construire une dérivation du triplet $\{A\}c\{A'\}$?
2. On souhaite répondre à la même question pour les *commandes* Imp *annotées*, décrits par la grammaire suivante :

   $c \ ::= \ \texttt{skip} \mid X := a \mid c_1; X := a \mid c_1; \{A\}c_2 \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } b \texttt{ do } \{A\}c$

   Ci-dessus, dans le cas $c_1; \{A\}c_2$, on impose que c_2 ne soit pas une séquence d'affectations. Dans le cas `while`, $\{A\}$ est l'invariant de boucle.
   L'idée est que l'utilisateur doit aider l'outil de preuve de programmes en ajoutant des assertions au code, aux endroits "stratégiques" [1].
   Définissez une fonction `VC`, qui associe à toute commande annotée un ensemble d'assertions de la forme $A_1 \Rightarrow A_2$. Ces assertions sont appelées des *verification conditions* (conditions de vérification), et correspondent à des utilisations de la règle de conséquence.

Dans la suite de cette séance, on s'intéresse à un outil mettant en œuvre les idées esquissées ci-dessus.

2 Why

Why est un outil qui permet certifier des programmes impératifs à l'aide d'annotations logiques. Le langage de programmation impératif correspond à la partie impérative de Caml. On peut aussi ajouter des annotations entre les séquences d'expressions. Voici un premier exemple :

```
module Swap
  use import int.Int
  use import ref.Ref
  let swap (a : ref int) (b : ref int)
    ensures { a = b.old /\ b = a.old }
    =
    let t = ref !a in
    a := !b;
    b := !t
end
```

C'est un programme qui échange le contenu de deux références. On comprend en lisant la post-condition que `a.old` désigne la valeur contenu dans `a` avant l'éxécution. On peut ajouter une pré-condition avec le mot-clef `requires` au lieu de `ensures`.

1. Il existe aussi des approches pour deviner automatiquement ces aides, y compris les invariants de boucle.

Préliminaires techniques Pour tester le système, dans une console :

1. tapez le fichier précédent dans un fichier `swap.mlw` puis :
2. `why3 prove swap.mlw -D coq > swap.v`
   (ceci génère les buts pour prouver la correction du programme dans un fichier `swap.v`)
3. `coqide -R /home/jmadiot/libwhy Why3 swap.v`
   (ceci ouvre `swap.v` dans coq avec la bibliothèque de Why3)

Question 3. Comprenez le but généré que Coq vous présente.

Question 4. Prouvez le but généré que Coq vous présente.

Question 5. Considérez le squelette de programme ci-dessous. Complétez-le dans un fichier `sort.mlw` de façon non-triviale et prouvez les obligations en Coq.

```
  let sort (a b c : ref int)
    ensures { !a <= !b /\ !b <= !c }
    =
    ...
```

Bien-sûr le programme

```
  b := a; c := b
```

satisfera bien sa post-condition. Pour spécifier plus précisément sont comportement, why nous permet d'axiomatiser des prédicats logiques. Ainsi on peut définit un nouveau prédicat inductif :

```
theory Permut
  use import int.Int
  inductive permut (int, int, int, int, int, int) =
    | Permut_init  : forall a b c. permut(a, b, c, a, b, c)
    | Permut_? : ...
    ...
end
```

Question 6. Recopiez et complétez l'axiomatisation pour que `permut(a, b, c, d, e, f)` signifie que $[a, b, c]$ est une permutation de $[d, e, f]$. Ensuite, remplacez la post-condition en utilisant `Permut.permut` et prouvez les nouvelles obligations.

Question 7. Prouvez la correction (partielle d'abord (enlevez le `variant`), puis totale) du programme suivant.

```
module Sum
  use import ref.Ref
  use import int.Int
  use import bool.Bool

  let sum (n : int)
    requires { 0 <= n }
    ensures  { 2 * result = n * (n + 1) }
    =
    let r = ref 0 in
    let k = ref 0 in
```

```
    while !k <= n do
      invariant { ... }
      variant { ... }
      r := !r + !k;
      k := !k + 1
    done;
    !r
end
```

2.1 Tactiques Coq utiles

Voici quelques tactiques Coq pour vous aider :

Anneaux `ring_simplify` simplifie le but en forme canonique pour les anneaux (par exemple $\mathbb{Z}$ avec multiplication et addition).
`ring_simplify in H` fait la même chose dans une hypothèse

Arithmétique `Require Import Omega` charge la tactique `omega` qui permet de résoudre les buts arithmétiques (par exemple `<=`) prouvables (sauf quand il faut manipuler les multiplications, pour ça, utilisez `ring_simplify`).

Prolog-like `Hint Constructors foo` charge les constructeurs de l'object inductif `foo`. Ensuite, on peut lancer les tactiques `auto` et `eauto` qui sont plus intelligentes car elles connaissent maintenant les constructeurs `foo`.

Égalités Vous pouvez utiliser `subst` quand vous avez une hypothèse de la forme `x = expr` pour remplacer `x` par `expr` partout.

Vous pouvez utiliser `injection H` quand vous avez une hypothèse de la forme `H: mk_ref a = mk_ref b` pour prouver qu'en fait, `a = b`. On peut automatiser le processus en programmant une tactique :

```
Ltac autoinj :=
  match goal with
  H: ?c ?a = ?c ?b |- _ =>
    injection H; clear H; intro H
  end.
```

on appelle cette tactique en faisant `autoinj` ou encore `repeat autoinj` pour l'appliquer tant que c'est possible.