TD Feuille 3 — Preuve formelle

Exercice 1 (Inférences formelles). On manipule ici rigoureusement le formalisme des règles d'inférence de Hoare sur l'IR des commandes gardées vue en CM.

▹ **Question 1.** Avec les règles (y compris dérivées) vues en cours, montrer la règle dérivée de calcul de WP ci-dessous. C'est la règle qu'on utilise quand on cherche à inférer l'invariant de boucle à partir de la postcondition.

$$\text{D}\ \frac{\{I \wedge C\}\, S\, \{I\}}{\{I\}\ \textbf{while}\ C\ \textbf{do}\ S\, \{Q\}}\ I \wedge \neg C \Rightarrow Q$$

On combine juste la règle dérivée du while avec la règle de conséquence.

$$\frac{\dfrac{\{I \wedge C\}\, S\, \{I\}}{\{I\}\ \textbf{while}\ C\ \textbf{do}\ S\, \{I \wedge \neg C\}}}{\{I\}\ \textbf{while}\ C\ \textbf{do}\ S\, \{Q\}}\ I \wedge \neg C \Rightarrow Q$$

On considère la définition de fonction suivante dans l'IR des commandes gardées.

$$\begin{array}{l}\textbf{pr}\ \mathrm{rem}(x, \texttt{\&mut}\ y)[\\ \qquad x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x \neq 0,\\ \qquad y = @y \mod x,\\ \qquad \textbf{while}\ x \leq y\ \textbf{do}\ y := y - x\\ ]\end{array}$$

On va montrer que cette définition est valide. Autrement dit, d'après le cours, il faut montrer ce lemme :

$$\text{D}\ \frac{}{\{@x \in \mathbb{N} \wedge @y \in \mathbb{N} \wedge @x \neq 0\}\,(x := @x; y := @y); \textbf{while}\ x \leq y\ \textbf{do}\ y := y - x\, \{y = @y \mod @x\}}$$

▹ **Question 2.** On commence par supposer qu'on a un invariant I. Construire un arbre de preuve complet (en propagation par WP) avec les obligations de preuves sur I. Pour abréger la construction de l'arbre, on note P (resp. Q) la précondition (resp. postcondition) du triplet de Hoare ci-dessus. Et on note W la boucle. Quand on pourra parcourir certains sous-arbres en propagation arrière (WP), on appliquera la technique vue à l'exo 2 de la Feuille 2 pour laisser ces sous-arbres implicites. A la fin, lister les obligations de preuve en fonction de I, avec leur noms usuels.

$$\frac{\dfrac{\dagger}{\{P\}\, x := @x; y := @y\, \{I\}}\ \text{INIT} \qquad \dfrac{\dfrac{}{\{I[y \leftarrow y - x]\}\, y := y - x\, \{I\}}\ \text{PRES}}{\{I \wedge x \leq y\}\, y := y - x\, \{I\}}\ \text{POST}}{\{I\}\, W\, \{Q\}}$$

$$\{P\}\, (x := @x; y := @y) W\, \{Q\}$$

où † est le sous-arbre

$$\mathrm{D}\ \frac{}{\{I[y \leftarrow @y][x \leftarrow @x]\}\, x{:=}@x; y{:=}@y\, \{I\}}$$

produit par la propagation arrière suivante

$-- \ I[y \leftarrow @y][x \leftarrow @x]$
$x{:=}@x; \ -- \ I[y \leftarrow @y]$
$y{:=}@y \ -- \ I$

et

INIT est l'initialisation de l'invariant $P \ \Rightarrow \ I[y \leftarrow @y][x \leftarrow @x]$
c-à-d. $\quad @x \in \mathbb{N} \wedge @y \in \mathbb{N} \wedge @x \neq 0 \ \Rightarrow \ I[y \leftarrow @y][x \leftarrow @x]$
PRES est la préservation de l'invariant $I \wedge x \leq y \ \Rightarrow \ I[y \leftarrow y - x]$
POST établit la postcondition $I \wedge \neg x \leq y \ \Rightarrow \ Q$
c-à-d. $\quad I \wedge y < x \ \Rightarrow \ y = @y \bmod @x$

On peut aussi construire l'arbre †, mais ça n'a pas grand intérêt :

$$\frac{\dfrac{}{\{I[y \leftarrow @y][x \leftarrow @x]\}\, x{:=}@x\, \{I[y \leftarrow @y]\}} \qquad \dfrac{}{\{I[y \leftarrow @y]\}\, y{:=}@y\, \{I\}}}{\{I[y \leftarrow @y][x \leftarrow @x]\}\, x{:=}@x; y{:=}@y\, \{I\}}$$

▹ **Question 3.** Il faut maintenant trouver le I sur lequel on peut prouver les obligations de preuves précédentes.

Indication à donner pour débloquer

on peut déjà remarquer que la précondition P ne porte que sur des variables logiques $@x$ et $@y$ qui ne sont pas des variables du programme : les conditions syntaxiques sur le corps des définitions de procédures garantissent que c'est toujours le cas ! Trivialement P est donc un invariant de boucle.

Ensuite, il faut identifier les variables de programmes qui ne sont pas modifiées par le corps de la boucle. Cela doit être exprimé dans un invariant de boucle. Enfin, le plus dur : identifier comment le résultat qu'on calcule (en postcondition) est préservé dans la boucle. L'invariant I est une généralisation de la postcondition.

Solution On prend $I \stackrel{def}{=} P \ \wedge \ x = @x \ \wedge \ (y \bmod @x) = (@y \bmod @x)$.

POST se déduit de $@x \neq 0 \wedge y < @x \ \Rightarrow \ (y \bmod @x) = y$.

PRES se déduit de $@x \neq 0 \wedge @x \leq y \ \Rightarrow \ (y \bmod @x) = ((y - @x) \bmod @x)$.

INIT se déduit de 2 égalités prouvées par réflexivité.

Info à donner dans les invariants de boucle Frama-C, on n'a pas besoin d'expliciter que la précondition P est un invariant. L'outil propage cette information pour nous.

▹ **Question 4.** En utilisant la commande gardée de l'appel de procédure, montrer

$$\mathrm{D}\ \frac{}{\{x = 19\}\, \mathrm{rem}(x - 15, \texttt{\&mut}\ x)\, \{x = 3\}}$$

On pourra abréger la construction de l'arbre en combinant propagation avant et arrière.

La commande (après substitution des paramètres) générée par l'appel est donnée ci-dessous (cf. le cours). Ici, en partant de la précondition, on propage en avant pour prouver le **assert** (qui est trivalement vrai car $x - 15 = 4$). Puis, on passe la précondition sous le **var** en ligne 3. Maintenant, en partant en arrière depuis la postcondition, on trouve que la plus faible précondition en ligne 4 est "$x \bmod (x-15) = 3$". Sous la précondition $x = 19$, ça se ramène à $19 \bmod 4 = 3$ (ce qui est vrai).

```
1 -- x = 19
2 assert (x − 15 ∈ ℕ ∧ x ∈ ℕ ∧ x − 15 ≠ 0); -- assert ok et x = 19
3 var y_f ( -- x = 19
4   -- y_f = x mod (x − 15) ⇒ y_f = 3 simplifiable en x mod (x − 15) = 3
5   assume y_f = x mod (x − 15); -- y_f = 3
6   x:=y_f -- x = 3
7 ) -- x = 3
```

Exercice 2 (Recherche linéaire dans un tableau). On considère le programme en figure 1, spécifié en Frama-C. Le tableau est ici donné sous la forme d'un pointeur (qui peut être alloué dynamiquement) et dont la taille n'est pas connu statiquement. Par rapport au schéma vu en cours, un accès en lecture à `t[i]` produit un assert RTE "`assert(\valid_read(t+i))`" (dans une mémoire implicite qui sera explicitée au moment du codage dans l'IR de vérification). Et la précondition sur le tableau est équivalente à

$$\forall i, 0 \leq i \leq \texttt{size} - 1 \Rightarrow \texttt{\valid\_read}(t + i)$$

La présence des 3 postconditions correspond au fait qu'elles sont établies en conjonction.

```
1  /*@ requires \valid_read(t+(0..size-1)) ;
2    @ assigns \nothing;
3    @ ensures -1 <= \result;
4    @ ensures 0 <= \result ==> \result < size && t[\result] == v;
5    @ ensures \result == -1 ==> \forall int k; 0 <= k < size ==> t[k] != v; */
6  int linear_search(int v, int* t, int size) {
7    int i ;
8    i=size-1;
9    while (i >= 0 && t[i] != v) i--;
10   return i;
11 }
```

Figure 1 – Recherche linéaire à compléter pour la preuve en Frama-C

▹ **Question 1.** Ajouter les assertions RTE.

Aïe difficulté ! On aurait envie d'ajouter des **assert** au milieu d'une condition, ce qui n'est pas possible dans notre formalisme. Frama-C évite le problème en transformant au préalable les boucles (cf. en TP). On ne va pas faire ça ici pour simplifier nos raisonnements sur la

boucle (qui restent en fait les mêmes que dans la version transformée par Frama-C, c'est juste pas évident à voir). Nos assertions RTE sont légèrement plus complexes que Frama-C par contre.

```
  /*@ assert(-2147483648 <= size-1); */
  i=size-1 ;
  /*@ assert(i >= 0 ==> \valid_read(t + i)); */
  while (i >= 0 && t[i] != v) {
     /*@ assert(-2147483648 <= i-1); */
     i--;
     /*@ assert(i >= 0 ==> \valid_read(t + i)); */
  }
  return i;
}
```

▹ **Question 2.** On veut prouver les assertions RTE - sans chercher à prouver forcément les postconditions pour l'instant, sauf si ça simplifie la vie. Modifier le programme si besoin (la spécification, elle, ne doit pas être changée) et trouver l'invariant de boucle Frama-C qui permet de prouver ces assertions.

De façon évidente le premier **assert** est faux en l'absence de précondition sur `size`. Comme la postcondition suggère `-1 <= i`, ça suggère d'ajouter un test qui élimine le cas `size <= 0`. Et là c'est bon.

```
  if (size <= 0) return -1;
  /*@ assert(-2147483648 <= size-1); */
  i=size-1 ;
```

Pour l'invariant de boucle, voilà les annotations qu'il faut fournir. Il ne faut pas oublier le `loop assigns` qui exprime l'invariant de boucle que toutes les variables sauf `i` restent à leur valeur d'avant la boucle. Si on ne la met pas, le prouveur ne sait pas que `size` n'est pas modifié dans le corps de boucle.

```
  /*@ loop invariant -1 <= i < size;
    @ loop assigns i ;
```

Avec ça, toutes les assertions RTE sont déchargées !

▹ **Question 3.** Compléter l'invariant afin de pouvoir établir les postconditions. On prouvera que la formule proposée est bien un invariant et on explicitera comment les 3 postconditions sont alors établies.

On a juste besoin d'ajouter l'invariant ci-dessous qui exprime que `v` n'est pas dans la tranche du tableau qui a été examinée jusqu'à `i` (exclus). La preuve de cet invariant est triviale.

```
   /* @ loop invariant \forall int k; i < k < size ==> t[k] != v ; */
```

La première postcondition dérive directement des invariants de boucle pour RTE (question précédentes). En sortie, la condition de boucle est fausse, avec soit `i >= 0`, soit `i < 0` mais

alors d'après l'invariant RTE, `i==-1`. Dans le premier cas, on doit prouver la deuxième postcondition : elle découle alors directement du fait que l'expression droite de la condition est fausse ; on a trouvé `v`. Dans le deuxième cas, on doit prouver la troisième postcondition qui dérive alors directement de l'invariant ci-dessus.

Exercice 3 (Théorie des tableaux en SMT-solving). On étudie comment raisonner sur les tableaux dans la logique. Ici, on ne s'occupe pas de la question débordements de tableaux, gérée au niveau des **assert** RTE, comme on l'a vu au TD précédent. On se base sur la théorie des tableaux définie par les 3 axiomes suivants :

$$\begin{aligned}
&\text{A1}: \quad \forall\, t,i,j,x,\ (i = j \Rightarrow\ \texttt{select}(\texttt{store}(t,i,x),j) = x)\\
&\text{A2}: \quad \forall\, t,i,j,x,\ (i \neq j \Rightarrow\ \texttt{select}(\texttt{store}(t,i,x),j) = \texttt{select}(t,j))\\
&\text{A3}: \quad \forall\, t_1,t_2,\ (\forall\, i,\ (\texttt{select}(t_1,i) = \texttt{select}(t_2,i)) \Rightarrow\ t_1 = t_2
\end{aligned}$$

▹ **Question 1.** La valeur d'un tableau est donc représenté comme une suite de `store`. En partant de la valeur initiale t_0 donner :

— la valeur t de `t` après la suite d'affectations C "`t[2]=0; t[5]=0; t[2]=1;`"

— le calcul des valeurs `t[2]` et `t[5]` pour le t calculé précédemment.

— Pour clarifier, on note t_1 et t_2 les tableaux intermédiaires (principe du Single-Static-Assignment).

$t_1 := \texttt{store}(t_0, 2, 0)$;
$t_2 := \texttt{store}(t_1, 5, 0)$;
$t := \texttt{store}(t_2, 2, 1)$;

En éliminant les variables intermédiaires, on trouve

$$t = \texttt{store}(\texttt{store}(\texttt{store}(t_0,2,0),5,0),2,1)$$

Mais pour la suite, on a intérêt à conserver ces variables intermédiaires.

— `t[2]` correspond à

$$\begin{aligned}
\texttt{select}(t,2) &= \texttt{select}(\texttt{store}(t_2,2,1),2)\\
&= 1 && \text{par A1}
\end{aligned}$$

— `t[5]` correspond à

$$\begin{aligned}
\texttt{select}(t,5) &= \texttt{select}(\texttt{store}(t_2,2,1),5)\\
&= \texttt{select}(t_2,5) && \text{par A2}\\
&= \texttt{select}(\texttt{store}(t_1,5,0),5)\\
&= 1 && \text{par A1}
\end{aligned}$$

▹ **Question 2.** On définit l'échange de 2 éléments d'un tableau par :

$$swap(t, i, j) \triangleq \texttt{store}(\texttt{store}(t, i, \texttt{select}(t, j)), j, \texttt{select}(t, i))$$

Prouver la propriété suivante : $\forall k,\ swap(t, k, k) = t$

Ci-dessous, je note `select`(t, i) par $t[i]$ et `store`(t, i, x) par $t[i \leftarrow x]$.

On peut utiliser la tactique de preuve suivante prouver les buts de la forme "`t1=t2`" : par A3 il suffit de prouver "`t1[n]=t2[n]`" avec un indice `n` qui n'apparaît pas dans le contexte ; ensuite, on applique A1 ou A2 en raisonnant par cas sur les indices qui apparaissent dans "`t1`" et "`t2`". Exemple pour `swap_refl`.

```
swap(t,k,k) =  t[k <- t[k]][k <- t[k]]

si n<>k alors swap(t,k,k)[n] = t[k <- t[k]][n] = t[n]
              (on applique 2 fois A2).
sinon, n=k et swap(t,k,k)[n] = t[k] = t[n]
      (on appplique 1 fois A1 + 1 fois le schéma de congruence).
```

La même tactique fonctionne pour tous les autres buts.

▹ **Question 3.** En supposant qu'une théorie associée au type **int** définisse un ordre total $\leq$, proposer une définition du prédicat `sorted(t,l,h)` qui exprime si t est trié par ordre croissant sur les indices compris dans `l..h`

On peut penser à 2 définitions qui sont équivalentes si on a l'induction des entiers (ci-dessous en syntaxe why3, pas présentée ici - car logique du 1er ordre typée hors programme). Vous pouvez juste oublier les types...

```
axiom weakly_sorted: forall a: (int,int) farray. forall l,h:int.
 sorted(a,l,h) <-> (forall i: int. l < i <= h -> a[i-1] <= a[i])

axiom strongly_sorted: forall a: (int,int) farray. forall l,h:int.
 sorted(a,l,h) <-> (forall i,j: int. l <= i <= j <= h -> a[i] <= a[j])
```

Malheureusement, on ne sait pas bien automatiser les preuves par induction (c'est indécidable). Donc les SMT solveurs n'ont généralement pas l'induction dans leur théorie des entiers. Et ils ne savent pas prouver l'équivalence entre les 2 définitions. Si besoin, on peut donc ajouter cette équivalence elle-même comme axiome.

▹ **Question 4.** Proposer une axiomatisation du prédicat $Permut(t_1, t_2)$ signifiant que t_2 est une permutation de t_1.

REM : même remarque que précédemment, en l'absence d'induction on peut éventuellement penser à plusieurs théories non équivalentes !

```
axiom permut_swap: forall t: ('a,'b) farray. forall i,j: 'a.
  permut(t,swap(t,i,j))

axiom permut_refl: forall t1,t2: ('a,'b) farray.
  t1=t2 -> permut(t1,t2)

axiom permut_sym: forall t1,t2: ('a,'b) farray.
  permut(t1,t2) -> permut(t2,t1)

axiom permut_trans: forall t1,t2,t3: ('a,'b) farray.
  permut(t1,t2) -> permut(t2,t3) -> permut(t1,t3)
```

Rem : l'axiome `permut_refl` est prouvable à partir de `permut_swap` via le résultat de la question 2. Ceci dit, ça aide le SMT-solveur de lui donner cet axiome plutôt que de lui laisser le réinférer par lui-même (cf. l'exercice `min_sort.c` du TP Frama-C).

▹ **Question 5.** Proposer une postcondition exprimant le tri du tableau dans une variable `t` de type `int t[100]`. On notera @*t* la valeur initiale logique du tableau `t`.

```
permut(@t,t) and sorted(t,0,99)
```