Logique de Hoare et preuves de programmes

Pierre Wilke
pierre.wilke@centralesupelec.fr

29 octobre 2018

Lors de ce TP, nous allons étudier la sémantique des programmes IMP, vue en cours. On présentera le langage, on définira sa sémantique en Coq, puis on prouvera les règles de la logique de Hoare, qui nous permettront d'effectuer la preuve fonctionnelle de deux programmes.

1 Le langage IMP

Nous utiliserons le langage IMP, vu en cours et dont la syntaxe et la sémantique sont rappelées ci-dessous. Les expressions manipulées sont les mêmes que lors du TP précédent. [1] On définit, en plus des expressions arithmétiques du TP précédents, des *conditions* (des expressions booléennes) `cond`, dont la syntaxe est donnée ci-dessous.

```
Inductive cond : Set :=
| Eq (e1 e2: expr)  (* e1 == e2 *)
| Lt (e1 e2: expr)  (* e1 <  e2 *)
| And (c1 c2: cond) (* c1 && c2 *)
| Or (c1 c2: cond)  (* c1 || c2 *)
| Not (c: cond).    (* ! c *)
```

Les instructions (*statements*) du langage IMP sont définies par le type Coq `stmt`, donné ci-dessous.

```
Inductive stmt : Set :=
| Skip                          (* ne rien faire *)
| Assign (v: var) (e: expr)     (* v := e *)
| Seq (s1 s2: stmt)             (* s1 ; s2 *)
| If (c: cond) (s1 s2: stmt)    (* if (c) { s1 } else { s2 } *)
| While (c: cond) (s: stmt).    (* while(c) { s } *)
```

1. On remarquera quand même deux différences mineures :
   a. les valeurs manipulées sont maintenant des entiers naturels plutôt que relatifs, et ce afin de faciliter quelques preuves ;
   b. les identifiants de variables sont maintenant des chaînes de caractère (`string`) plutôt que des entiers naturels, ce qui facilite la lecture et l'écriture de ces programmes.

Les sémantiques des expressions, des conditions et des instructions sont données par la fonction `eval_expr`, la fonction `eval_cond` et la relation `eval_stmt` respectivement, définies dans le squelette de ce TP. Pour la sémantique des instructions, les règles correspondant aux constructions `If` et `While` sont manquantes.

La sémantique des instructions, telle que donnée en cours, est rappelée ci-dessous.

$$\frac{}{(\texttt{skip}, env) \to env} \qquad \frac{}{(x{:=}e, env) \to \texttt{update\_env}\ env\ x\ (\texttt{eval\_expr}\ env\ e)}$$

$$\frac{(s_1, env) \to env' \qquad (s_2, env') \to env''}{(s_1; s_2, env) \to env''} \qquad \frac{\texttt{eval\_expr}\ env\ e = 0}{(\texttt{while}(e)\{s\}, env) \to env}$$

$$\frac{\texttt{eval\_expr}\ env\ e \neq 0 \qquad (s, env) \to env' \qquad (\texttt{while}(e)\{s\}, env') \to env''}{(\texttt{while}(e)\{s\}, env) \to env''}$$

$$\frac{\texttt{eval\_expr}\ env\ e \neq 0 \quad (s_1, env) \to env'}{(\texttt{if}(e)\ \texttt{then}\ s1\ \texttt{else}\ s2, env) \to env'} \qquad \frac{\texttt{eval\_expr}\ env\ e = 0 \quad (s_2, env) \to env'}{(\texttt{if}(e)\ \texttt{then}\ s1\ \texttt{else}\ s2, env) \to env'}$$

Question 1.1 *Compléter la relation `eval_stmt` pour les 4 règles correspondant aux constructions `If` et `While`.*

2 Quelques détails sur des tactiques utiles en Coq

2.1 Manipulation de relations en Coq.

Coq fournit des outils pour manipuler des relations.

Lorsqu'une hypothèse a pour type une relation, par exemple :

```
H : eval_stmt env (Assign v e) env'
```

on peut utiliser la tactique `inv H`, qui va générer un sous-but par constructeur de la relation `eval_stmt` compatible avec `Assign v e`, et dans chaque tel sous-but, l'environnement `env'` est remplacé par l'environnement décrit dans le constructeur.

Dans l'autre sens, c'est-à-dire pour prouver une relation (notre but est une relation), on a plusieurs techniques :

- on peut donner le nom du constructeur approprié et utiliser la tactique `apply constructeur` ;
- on peut laisser Coq trouver le constructeur qui correspond, et utiliser la tactique `constructor`. Attention, plusieurs constructeurs peuvent être compatibles avec le but : dans ce cas, Coq choisira le premier qu'il trouve (pas forcément celui que l'on voulait appliquer) ;

— dans certains cas, l'application du constructeur (que ce soit *via* `apply` ou *via* `constructor`) peut nécessiter que l'on fournisse des informations supplémentaires que Coq seul ne peut pas inventer. Par exemple, le constructeur `eval_seq` (dans `eval_stmt`) a besoin d'un environnement intermédiaire `env2`, que Coq ne peut pas inférer (déduire) à partir du but. Dans ce cas, on a deux solutions :

  — soit on donne le(s) paramètre(s) manquant(s) avec la syntaxe `with` :

  ```
  apply eval_seq with (env2 := ...).
  ```

  Cela nécessite d'écrire à la main l'environnement, ce qui peut être fastidieux et propice à introduire des erreurs

  — soit, on utilise les tactiques `eapply` ou `econstructor` qui vont générer des *variables existentielles*. Ces variables existentielles devront être instanciées plus tard dans la preuve. Un exemple d'utilisation de ces tactiques est donné dans le squelette de la preuve (`eval_2_assign`).

2.2 La tactique `induction`

Quelques remarques sur la tactique `induction`.
Lorsque le but de la preuve est de la forme

```
f a b = x
```

avec `a` de type `nat` (par exemple), la tactique `induction a`, génère les deux sous-buts suivants :

1. `f 0 b = x`
2. `IHa : f a b = x ⊢ f (S a) b = x`

On remarque que l'hypothèse d'induction `IHa` est de la forme `f a b = x` et n'est donc valable que pour ce `b` donné et pas pour tout `b`. Pour avoir une hypothèse d'induction plus générale (*i.e.* qui est valable pour tout `b`), il aurait fallu ne pas introduire `b` dans le contexte (avec `intros`) avant de lancer `induction a`, ou bien de faire revenir `b` dans le but grâce à la tactique `revert b`. Ainsi, si le but est de la forme `forall b x, f a b = x`, la tactique `induction a` génère ces deux sous-buts :

1. `forall b x, f 0 b = x`
2. `IHa: forall b x, f a b = x ⊢ forall b x, f (S a) b = x`

2.3 Déplier des définitions

On peut remplacer une fonction par sa définition grâce à la tactique `unfold`. Par exemple, admettons que l'on ait défini une fonction comme cela :

```
Definition f (x: nat) := x * x + 2.
```

Si on a un but de la forme `f a + f b = f c`, alors la tactique `unfold f` remplacera le but par `a * a + 2 + b * b + 2 = c * c + 2`.

Attention : `unfold` ne fonctionne que sur des termes définis avec `Definition`. Les définitions `Inductive` ne peuvent pas être « unfoldées ». Ce n'est en général pas une bonne idée d'appliquer `unfold` sur des fonctions récursives (`Fixpoint`).

3 Logique de Hoare

On veut maintenant définir la logique de Hoare dans Coq afin de pouvoir raisonner sur le comportement de programmes IMP. On rappelle que la logique de Hoare s'intéresse à des triplets $\{P\}s\{Q\}$ où P et Q sont des prédicats sur l'environnement et s est une instruction (un *statement*).

On définit le type `pred` des prédicats comme ci-dessous.

Un triplet de Hoare $\{P\}s\{Q\}$ est valide si pour tout environnement env_1 qui satisfait la précondition P, tout environnement env_2 qui est le résultat de l'évaluation de s satisfait Q.

```
Definition pred := envt -> Prop.

Definition valid_hoare_triple (P: pred) (s: stmt) (Q: pred) : Prop :=
  forall env1 env2,
    P env1 ->
    eval_stmt env1 s env2 ->
    Q env2.
```

3.1 Règles de la logique de Hoare

On rappelle l'ensemble des règles de la logique de Hoare ci-dessous.

$$\text{SKIP}\ \frac{}{\{P\}\texttt{Skip}\{P\}} \qquad \text{WEAKEN}\ \frac{\{P\}s\{Q\} \qquad Q \Rightarrow Q'}{\{P\}s\{Q'\}} \qquad \text{STRENGHTEN}\ \frac{P' \Rightarrow P \qquad \{P\}s\{Q\}}{\{P'\}s\{Q\}}$$

$$\text{SÉQUENCE}\ \frac{\{P\}s_1\{Q\} \qquad \{Q\}s_2\{R\}}{\{P\}s_1; s_2\{R\}} \qquad \text{WHILE}\ \frac{\{P \wedge e\}s\{P\}}{\{P\}\ \texttt{while}(e)\{s\}\ \{P \wedge \neg e\}}$$

$$\text{AFFECTATION}\ \frac{}{\{P[x \leftarrow E]\}\ x{:=}E\ \{P\}} \qquad \text{CONDITION}\ \frac{\{P \wedge e\}s_1\{Q\} \qquad \{P \wedge \neg e\}s_2\{Q\}}{\{P\}\texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2\{Q\}}$$

Un squelette est fourni pour toutes ces règles. Dans certains cas, la preuve est entièrement faite. Dans d'autres, elle n'est que partiellement commencée, et vous devez compléter la suite. Dans d'autres encore, il vous reste toute la preuve à faire.

Question 3.1 *Compléter les preuves des règles* ***`hoare_seq`****,* ***`hoare_if`****,* ***`hoare_while`****,* ***`hoare_assign`****,* ***`hoare_strengthen_pre`****,* ***`hoare_weaken_post`****.*

L'utilisation de la règle `WHILE` nécessitera souvent de généraliser la précondition et la postcondition. Il peut être intéressant de prouver la règle `WHILE'` qui compose les règles `WEAKEN`, `WHILE` et `STRENGTHEN`. Cela simplifie l'utilisation de cette règle, en ne nécessitant de fournir que l'invariant `I`.

$$\text{WHILE'}\quad \frac{P \Rightarrow I \qquad \{I \wedge e\}s\{I\} \qquad I \wedge \neg e \Rightarrow Q}{\{P\}\ \texttt{while}(e)\{s\}\ \{Q\}}$$

Question 3.2 *Prouver le lemme* `hoare_while'`.

3.2 Plus faible précondition (*Weakest precondition*)

Pour les programmes sans boucles, il est possible de déduire automatiquement la précondition la plus faible pour une post-condition donnée. En d'autres termes, si on a un programme s et une post-condition Q que l'on souhaite satisfaire, on peut calculer une précondition $P = wp(s, Q)$ telle que $\{P\}s\{Q\}$ est un triplet de Hoare valide (et P n'est pas *trop* restrictif ; sinon `False` ferait très bien l'affaire !).

Question 3.3 *Écrire un prédicat* `no_whiles` *qui prend une instruction et détermine si l'instruction contient une instruction* `While c s` *ou pas. La signature de ce prédicat sera la suivante :*

```
Fixpoint no_whiles (s: stmt) : Prop :=
```

Note : on définit ici un prédicat plutôt qu'une fonction booléenne. On n'utilisera donc pas le « et » booléen (&&) mais le « et » logique (/\).

La fonction `wp`, qui prend une instruction et une postcondition et qui donne la précondition la plus faible associée, vous est donnée.

```
Fixpoint wp (s: stmt) (Q: pred) : pred :=
 match s with
  | Skip => Q
  | Assign v e => fun env => Q (update_env env v (eval_expr env e))
  | Seq s1 s2 => wp s1 (wp s2 Q)
  | If c s1 s2 =>
    fun env => if eval_cond env c then wp s1 Q env else wp s2 Q env
  | While c s => Q
 end.
```

Note 1 : le *pattern-matching* en Coq doit être total, c'est-à-dire qu'on doit traiter tous les constructeurs. Dans notre cas, on ne souhaiterait pas traiter le cas `While` puisqu'on ne sait pas en donner une plus faible précondition. On peut renvoyer un prédicat arbitraire dans notre cas, puisqu'on ne se servira de `wp` que pour des programmes sans `While`.

Note 2 : la fonction `wp` renvoie un résultat de type `pred`, c'est-à-dire de type `envt -> Prop` : un prédicat sur les environnements. On peut en Coq construire une telle fonction grâce à la syntaxe suivante :

```
fun x => x + 1
```

On appelle une telle fonction une fonction *anonyme* puisqu'elle n'a pas de nom. On les appelle également des *lambda-fonctions* (ou λ-fonctions).
La signification de `fun x => x + 1` est : une fonction qui prend un paramètre (`x`) et qui retourne un résultat (`x + 1`). Par exemple, pour le cas `Assign`, `fun env => Q (update_env env v (eval_expr env e))`, est une fonction qui prend un environnement `env` en paramètre et dont le résultat est l'environnement `env` mis à jour pour que la variable `v` vaille la valeur de l'expression `e` dans l'environnement `env`.

On peut maintenant se servir de ce calcul de plus faible précondition pour simplifier le raisonnement sur des programmes (ou des morceaux de programme) sans boucles. On peut par exemple prouver le théorème suivant :

```
Theorem auto_hoare:
  forall s Q,
    no_whiles s ->
    valid_hoare_triple (wp s Q) s Q.
```

Autrement dit, pour tout programme s sans boucles, et pour tout prédicat Q, le triplet $\{wp(s, Q)\}s\{Q\}$ est valide.

Question 3.4 *Prouver le théorème `auto_hoare`.*

Indice : ce théorème se prouve par induction sur `s`. On pourra commencer la preuve par `induction s; simpl; intros Q NW`. Chaîner la tactique `induction` avec `simpl` et `intros` est généralement une bonne manière de commencer une preuve par induction, puisqu'on applique les tactiques de simplification et d'introduction à tous les sous-buts.

La plupart du temps, la précondition dans votre but ne coincidera pas exactement avec la précondition la plus faible. On pourra alors utiliser le lemme suivant, qui combine le théorème `auto_hoare` que vous venez de prouver, avec la règle de renforcement de la précondition (`hoare_strengthen_pre`) :

```
Lemma auto_hoare':
  forall (P: pred) s Q,
    no_whiles s ->
    (forall env, P env -> wp s Q env) ->
    valid_hoare_triple P s Q.
```

Question 3.5 *Prouver le lemme `auto_hoare'`.*

Dans le squelette, vous trouverez l'exemple `swap` qui fournit un exemple d'application de ce lemme pour prouver simplement que le programme échange les valeurs de deux variables. Étudiez cette preuve pour comprendre à la fois la spécification (l'énoncé du théorème) et comment le lemme `auto_hoare'` est utilisé.

4 Premières preuves de programme

On a désormais les outils nécessaires pour prouver des propriétés sur les programmes. Commençons par prouver en Coq le programme vu en fin de cours. Le programme est le suivant :

```
Definition premier_programme : stmt :=
  While (Lt (Var "x") (Var "b")) (Assign "x" (Add (Var "x") (Const 1))).
```

Et la propriété que l'on souhaite prouver est la suivante : si x est inférieur ou égal à b dans l'environnement initial, alors à la fin de l'exécution du programme, on a $x = b$. Voici la preuve que l'on a faite en cours :

$$\dfrac{\dfrac{\dfrac{\dots}{\{x \leq b \wedge x < b\}(\{x := x+1\})\ \{x \leq b\}}}{\{x \leq b\}(\texttt{while}\ (x < b)\{x := x+1\})\ \{x \leq b \wedge \neg(x < b)\}} \qquad \dfrac{\dots}{x \leq b \wedge \neg(x < b) \Rightarrow x = b}}{\{x \leq b\}(\texttt{while}\ (x < b)\{x := x+1\})\ \{x = b\}}$$

Question 4.1 *Reproduisez cette preuve en Coq.*

Note 1 : la preuve est grandement simplifiée par l'utilisation des lemmes `hoare_while'` *et* `auto_hoare'`.

Note 2 : Vous aurez besoin des lemmes suivants, qui font la conversion entre les comparaisons `Nat.ltb : nat -> nat -> bool` *et* `Nat.lt : nat -> nat -> Prop` *(notée <).*

```
Nat.ltb_lt : forall n m : nat, (n <? m) = true <-> n < m
Nat.ltb_nlt : forall x y : nat, (x <? y) = false <-> ~ x < y
le_nlt_eq: forall a b: nat, a <= b -> ~ a < b -> a = b.
```

Note 3 : Coq fournit la tactique `lia` *qui résoud les buts d'arithmétique sur les entiers naturels.*

5 PGCD

On propose maintenant de vérifier un algorithme qui calcule le plus grand diviseur commun de deux entiers. L'algorithme est le suivant :

while $x \neq y$ **do**
  if $x < y$ **then**
    y ← y - x
  else
    x ← x - y
  end
end

À la fin de la boucle, les variables `x` et `y` sont égales au PGCD des valeurs initialement contenues dans ces variables. La correction de cet algorithme est établie à l'aide de ces deux théorèmes :

```
Nat.gcd_sub_diag_r :
 forall n m : nat, n <= m -> Nat.gcd n (m - n) = Nat.gcd n m
Nat.gcd_comm : forall n m : nat, Nat.gcd n m = Nat.gcd m n
Nat.gcd_diag : forall n : nat, Nat.gcd n n = n
```

Le cœur de la preuve correspond à trouver l'invariant de la boucle (*i.e.* le `P` dans le lemme `hoare_while`).

Question 5.1 *Trouver l'invariant de la boucle (sur papier).*

Question 5.2 *Prouver le théorème `gcd_correct`. Vous aurez besoin des lemmes suivants :*

```
beq_nat_false : forall n m : nat, (n =? m) = false -> n <> m
beq_nat_true : forall n m : nat, (n =? m) = true -> n = m
Bool.negb_true_iff : forall b : bool, negb b = true <-> b = false
Bool.negb_false_iff : forall b : bool, negb b = false <-> b = true
```

6 Factorielle

De la même manière, on s'intéresse maintenant au programme suivant, qui calcule la factorielle d'un nombre.

L'algorithme est le suivant :

res ← 1
while $n \neq 0$ **do**
  res ← res * n
  n ← n - 1
end

Question 6.1 *Écrire le programme `factorielle`.*

Question 6.2 *Écrire la preuve que `factorielle` calcule la factorielle comme spécifié par la fonction `fact` définie en Coq sur les naturels. On se servira de la tactique `lia` pour les preuves arithmétiques et du lemme suivant, prouvé dans le squelette :*

```
Lemma fact_S:
  forall n,
    n <> 0 ->
    fact n = n * fact (n - 1).
```

Encore une fois, le cœur de la preuve est l'invariant de la boucle.

Lemmes utiles :

```
Nat.mul_1_r : forall n : nat, n * 1 = n
```