

TP 2 : Sémantique et logique de Hoare

Ce TP sera effectué sur deux séances : la première portant sur la notion de sémantique formelle, et la seconde sur la logique de Hoare.

L'objectif de la première séance de TP sera de définir deux sémantiques formelles pour le langage WHILE étudié en cours : une sémantique naturelle (à grands pas) et une sémantique à petits pas. Vous prouverez l'équivalence entre ces deux sémantiques. Vous construirez également un interpréteur du langage WHILE, et prouverez sa correction vis-à-vis de la sémantique naturelle. Les sémantiques que vous définirez seront utilisées dans le TP 3, qui portera sur l'analyse statique et la compilation vérifiée.

Lors de la seconde séance de TP, vous formaliserez les règles de la logique de Hoare vues en cours, et prouverez leur correction vis-à-vis de la sémantique à grands pas du langage WHILE.

Mise en place du TP

Pour vous mettre sur les rails, vous épargner l'écriture fastidieuse de définitions rébarbatives, et vous faciliter la vie, nous vous avons préparé un squelette de ce TP, qu'il ne vous restera qu'à remplir. Récupérez le squelette ici :

```
$ git clone git@gitlab-research.centralesupelec.fr:cidre-public/seculog.git
OU
$ git clone https://gitlab-research.centralesupelec.fr/cidre-public/seculog.git
```

Vous récupérez un répertoire avec les fichiers suivants :

- `_CoqProject` : il s'agit d'un fichier de configuration, à partir duquel on pourra générer un Makefile
- `While.v` : syntaxe du langage WHILE, évaluation des expressions et des conditions
- `WhileBigStep.v` : sémantique à grands pas
- `WhileSmallStep.v` : sémantique à petits pas
- `Equiv.v` : preuve d'équivalence entre les deux sémantiques
- `Interp.v` : définition de l'interpréteur et preuve de correction
- `Hoare.v` : règles de la logique de Hoare
- `UseHoare.v` : preuves de programme en utilisant la logique de Hoare

Tout d'abord, générez le Makefile avec la commande suivante :

```
$ coq_makefile -f _CoqProject -o Makefile
```

Puis construisez le projet en tapant `make`.

À chaque fois que vous faites une modification dans un fichier, pour que vos changements soient visibles depuis les autres fichiers, pensez à recompiler votre projet en tapant `make` dans le terminal.

C'est parti!

À titre indicatif, la première séance devrait idéalement couvrir les Sections 1 à 5, et la deuxième séance devrait couvrir les Sections 6 à 8.

1 Syntaxe du langage While et sémantique des expressions

Les fonctions et preuves que vous aurez à écrire dans cette section seront dans le fichier `While.v`.

Ce fichier contient les définitions du langage WHILE. Les variables sont identifiées par des chaînes de caractères. Les valeurs manipulées par les programmes sont des entiers naturels (`nat`). Le type des expressions est défini par le type `expr`, c'est le même que celui étudié en cours : des constantes (`Const n`), des variables (`Var x`), des additions, multiplications ou soustractions de deux expressions. Les conditions sont définies par le type `cond`.

L'état des programmes (type `state`) associe à chaque variable une valeur. Ce type est `var -> val`, i.e. des fonctions qui prennent un nom de variable et qui retournent une valeur. Un tel état peut être mis à jour grâce à la fonction `update_state`. Ce que l'on écrivait $\sigma[x \mapsto v]$ dans le cours devient `update_state env x v` en Coq. (On notera que la fonction `update_state` utilise la fonction `var_eq`, qui ne renvoie pas un booléen mais une preuve d'égalité ou d'inégalité entre deux noms de variables.)

La fonction d'évaluation des expressions n'est pas donnée en entier, c'est à vous de la compléter.

Activité 1.1. Complétez le code de la fonction `eval_expr`.

Activité 1.2. Complétez le code de la fonction `eval_cond : state -> cond -> bool`.

Il est souvent préférable, lorsque l'on prouve des théorèmes, de travailler avec des prédicats plutôt que des fonctions booléennes.

Activité 1.3.

1. Complétez le code du prédicat `eval_condP : state -> cond -> Prop`.
2. Prouvez ensuite, par induction sur `c`, le lemme suivant, qui relie les deux versions de la sémantique des conditions :

```
Lemma eval_cond_true:
  forall env c,
    eval_condP env c <->
    eval_cond env c = true.
```

3. Prouvez, en utilisant `eval_cond_true`, et sans utiliser `induction`, le lemme suivant :

```
Lemma eval_cond_false:
  forall env c,
    ~ eval_condP env c <->
    eval_cond env c = false.
```

Indice : vous pouvez utiliser `rewrite H` lorsque H est de type $A \leftrightarrow B$.

Vous pourrez utiliser les lemmes suivants pour effectuer votre preuve :

- `Z.eqb_eb`
- `Z.ltb_lt`
- `Bool.andb_true_iff`
- `Bool.orb_true_iff`

```
— Bool.negb_false_iff
— Bool.negb_true_iff
```

Dans le TP 1, nous avons utilisé la fonction `lt_dec` : `forall n m : nat, {n < m} + {~ n < m}`. Pour rappel, il s'agit d'une fonction prenant deux entiers naturels `n` et `m` en paramètre, et qui renvoie soit une preuve de `n < m`, soit une preuve de `~ n < m`. Ce genre de fonctions montre la décidabilité de la relation `<` sur les entiers naturels, *i.e.* l'existence d'un algorithme qui décide l'ordre sur les entiers naturels.

Activité 1.4. Écrivez la preuve du lemme `eval_cond_dec`, qui décide si l'évaluation d'une condition dans un état donné est vraie ou fausse.

```
Lemma eval_cond_dec:
  forall env c, {eval_condP env c} + {~ eval_condP env c}.
```

Vous finirez cette preuve avec le mot-clé `Defined`. plutôt que `Qed`. En effet, `Qed` rend le contenu de la preuve *opaque*, et il est impossible d'évaluer cette fonction avec `Compute` par la suite. Au contraire, `Defined` rend le contenu de la preuve *transparent* et permet l'évaluation. La commande `Compute` située après la définition de `eval_cond_dec` dans le squelette devrait afficher `= left eq_refl`. Si ce n'est pas le cas, quelque chose ne va pas dans votre définition de `eval_expr` ou `eval_condP`.

Finalement, le type `stmt` définit les instructions du langage WHILE. Le constructeur `While` a trois paramètres : une condition `c`, un invariant `I` et une instruction à exécuter `s` (le corps de la boucle). L'invariant ne sera pas utilisé avant la section sur la logique de Hoare. Il n'a aucun impact sur la sémantique des programmes.

2 Sémantique à grands pas

Intéressons-nous maintenant au fichier `WhileBigStep.v`.

Dans le cours, nous avons défini la sémantique à grands pas des instructions du langage WHILE, notée $(i, \sigma) \Downarrow \sigma'$, avec l'ensemble de règles d'inférence suivant.

$$\begin{array}{c}
\text{BIGSTEPSKIP} \frac{}{(\text{skip}, \sigma) \Downarrow \sigma} \qquad \text{BIGSTEPASSIGN} \frac{}{(\mathbf{x} := e, \sigma) \Downarrow \sigma[x \mapsto \llbracket e \rrbracket_\sigma]} \\
\\
\text{BIGSTEPSEQ} \frac{(i_1, \sigma) \Downarrow \sigma' \quad (i_2, \sigma') \Downarrow \sigma''}{(i_1 ; i_2, \sigma) \Downarrow \sigma''} \\
\\
\text{BIGSTEPWHILEFALSE} \frac{\llbracket c \rrbracket_\sigma = \text{false}}{(\mathbf{while} (c) \text{ do } i, \sigma) \Downarrow \sigma} \\
\\
\text{BIGSTEPWHILETRUE} \frac{\llbracket c \rrbracket_\sigma = \text{true} \quad (i, \sigma) \Downarrow \sigma' \quad (\mathbf{while} (c) \text{ do } i, \sigma') \Downarrow \sigma''}{(\mathbf{while} (c) \text{ do } i, \sigma) \Downarrow \sigma''} \\
\\
\text{BIGSTEPIFTTRUE} \frac{\llbracket c \rrbracket_\sigma = \text{true} \quad (i_1, \sigma) \Downarrow \sigma'}{(\mathbf{if} (c) \text{ then } i_1 \text{ else } i_2, \sigma) \Downarrow \sigma'} \\
\\
\text{BIGSTEPIFFALSE} \frac{\llbracket c \rrbracket_\sigma = \text{false} \quad (i_2, \sigma) \Downarrow \sigma'}{(\mathbf{if} (c) \text{ then } i_1 \text{ else } i_2, \sigma) \Downarrow \sigma'}
\end{array}$$

En Coq, nous définissons cette relation comme un prédicat inductif `bigstep` : `state -> stmt -> state -> Prop`. Ainsi, ce que l'on écrivait dans le cours $(i, \sigma) \Downarrow \sigma'$ s'écrit en Coq `bigstep st i st'`.

Chaque règle d'inférence correspond à un constructeur du type `bigstep`. Par exemple, la règle `BIGSTEPIFTTRUE` devient le constructeur :

```

| bigstep_if_true:
  forall env c s1 s2 env',
    eval_condP env c ->
    bigstep env s1 env' ->
    bigstep env (If c s1 s2) env'

```

Activité 2.1. Complétez le prédicat inductif `bigstep` en ajoutant les constructeurs manquants `bigstep_if_false`, `bigstep_while_true` et `bigstep_while_false`, correspondant aux règles `BIGSTEPIFFALSE`, `BIGSTEPWHILETRUE` et `BIGSTEPWHILEFALSE`.

Une première preuve utilisant la sémantique à grands pas vous est fournie, pour vous montrer l'utilisation de `apply`, `eapply`, `constructor` et `econstructor` avec des prédicats inductifs. Étudiez la preuve de `test_bigstep_seq_assign`. Cette preuve n'utilise que des constructeurs qui sont déjà présents dans le squelette, et devrait donc passer sans problèmes.

Une second preuve, `test_bigstep_while`, fait appel aux constructeurs que vous devez écrire.

Activité 2.2. Décommentez la preuve de `test_bigstep_while`, en l'adaptant au besoin si vous avez utilisé des noms différents pour les constructeurs, ou si l'ordre des hypothèses est différent de celui que nous avons imaginé... Cette preuve devrait passer. (Considérez cette preuve comme une sorte de test unitaire de votre prédicat `bigstep`).

3 Sémantique à petits pas

Concentrons-nous maintenant sur le fichier `WhileSmallStep.v`.

Bonne nouvelle, ce fichier est complet, vous n'avez rien à y ajouter.

On définit la sémantique à petits pas comme un prédicat inductif

`step : stmt -> state -> stmt -> state -> Prop.`

Ainsi, ce que l'on écrivait $(i, \sigma) \rightarrow (i', \sigma')$ dans le cours devient `step i env i' env'` en Coq.

Chacun des constructeurs correspond à une des règles d'inférence vues en cours, rappelées ci-dessous :

$$\begin{array}{c}
 \text{STEPASSIGN} \frac{}{(x := e, \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto \llbracket e \rrbracket_\sigma])} \\
 \\
 \text{STEPSEQ} \frac{(i_1, \sigma) \rightarrow (i'_1, \sigma')}{(i_1 ; i_2, \sigma) \rightarrow (i'_1 ; i_2, \sigma')} \qquad \text{STEPSEQSKIP} \frac{}{(\text{skip} ; i_2, \sigma) \rightarrow (i_2, \sigma')} \\
 \\
 \text{STEPIFTRUE} \frac{\llbracket b \rrbracket_\sigma = \text{true}}{(\text{if } (b) \text{ then } i_1 \text{ else } i_2, \sigma) \rightarrow (i_1, \sigma)} \\
 \\
 \text{STEPIFFALSE} \frac{\llbracket b \rrbracket_\sigma = \text{false}}{(\text{if } (b) \text{ then } i_1 \text{ else } i_2, \sigma) \rightarrow (i_2, \sigma')} \\
 \\
 \text{STEPWHILETRUE} \frac{\llbracket b \rrbracket_\sigma = \text{true}}{(\text{while } (b) \text{ do } i, \sigma) \rightarrow (i; \text{while } (b) \text{ do } i, \sigma)} \\
 \\
 \text{STEPWHILEFALSE} \frac{\llbracket b \rrbracket_\sigma = \text{false}}{(\text{while } (b) \text{ do } i, \sigma) \rightarrow (\text{skip}, \sigma')}
 \end{array}$$

Par exemple, le constructeur `step_while_true` correspond à la règle `STEPWHILETRUE` :

```

| step_while_true:
  forall env c I s,
    eval_condP env c ->
      step (While c I s) env (Seq s (While c I s)) env

```

Activité 3.1. Examinez chacun des constructeurs, pour vous convaincre qu'ils correspondent bien aux règles vues en cours.

On définit également le prédicat `star` qui correspond à 0 ou plusieurs étapes de `step`. Cela correspond aux règles suivantes, vues en cours :

$$\begin{array}{c}
 \text{STARREFL} \frac{}{(i, \sigma) \rightarrow^* (i, \sigma)} \\
 \\
 \text{STARSTEP} \frac{(i, \sigma) \rightarrow (i', \sigma') \quad (i', \sigma') \rightarrow^* (i'', \sigma'')}{(i, \sigma) \rightarrow^* (i'', \sigma'')}
 \end{array}$$

4 Équivalence entre les deux sémantiques

On s'intéresse dans cette section à prouver l'équivalence entre la sémantique à grands pas (**bigstep**) et la sémantique à petits pas (**star** et **step**). Plus précisément, on voudra prouver :

— $\forall i \sigma \sigma', (i, \sigma) \Downarrow \sigma' \Rightarrow (i, \sigma) \rightarrow^* (\text{skip}, \sigma')$

c'est-à-dire, le théorème Coq suivant :

```
Theorem bigstep_star: forall i env env',
  bigstep env i env' -> star i env Skip env'.
```

— $\forall i \sigma \sigma', (i, \sigma) \rightarrow^* (\text{skip}, \sigma') \Rightarrow (i, \sigma) \Downarrow \sigma'$

c'est-à-dire, le théorème Coq suivant :

```
Theorem star_bigstep: forall i env env',
  star i env Skip env' -> bigstep env i env'.
```

Le squelette de cette partie du TP se trouve dans le fichier `Equiv.v`.

Tout d'abord, nous aurons besoin de prouver le lemme suivant, qui exprime une propriété de \rightarrow^* vis-à-vis de la construction `Seq s1 s2` :

c'est-à-dire en Coq :

```
Lemma star_seq:
  forall s1 env1 env2, star s1 env1 Skip env2 ->
  forall s2 s3 env3, star s2 env2 s3 env3 ->
  star (Seq s1 s2) env1 s3 env3.
Proof.
  intros s1 env1 env2 Hstar.
  dependent induction Hstar; simpl; intros.
```

Il s'agit du début d'une preuve par induction sur l'hypothèse `star s1 env1 Skip env2`. La tactique `induction` directement appliquée sur cette hypothèse « oublierait » le fait que le troisième paramètre de `star` est `Skip`, et rendrait la preuve impossible. L'utilisation de la tactique `dependent induction`, puis puissante, permet d'éliminer ce problème¹.

Activité 4.1. Terminez la preuve de `star_seq`.

Vous devrez utiliser les constructeurs de `star` et `step`. Vous pouvez lancer `Print star.` et `Print step.` pour afficher l'ensemble des constructeurs de `star` et `step`.

Nous pouvons à présent prouver la première partie de l'équivalence entre les deux sémantiques :

Activité 4.2. Prouvez le théorème `bigstep_star` :

```
Theorem bigstep_star:
  forall i env env',
    bigstep env i env' ->
    star i env Skip env'.
```

Il s'agit d'une preuve par induction sur l'hypothèse `bigstep env i env'`. Vous devrez utiliser

1. Consultez <https://coq.inria.fr/refman/proof-engine/detailed-tactic-examples.html> pour plus de détails (mais plus technique...) sur cette tactique.

les constructeurs de **star** et **step**, ainsi que le lemme prouvé à l'activité précédente.

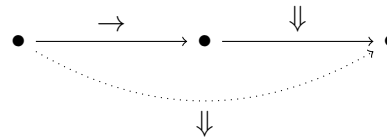
Intéressons-nous maintenant à l'autre partie de l'équivalence.

Avant de prouver le théorème final portant sur la relation \rightarrow^* (**star**), prouvons ce lemme portant sur \rightarrow (**step**) :

```

Lemma step_bigstep:
  forall i env i' env',
    step i env i' env' ->
    forall env'',
      bigstep env' i' env'' ->
      bigstep env i env''.

```



Ce lemme montre que faire un petit pas puis un grand pas est équivalent à faire un grand pas. Le schéma ci-dessus illustre ce lemme, où les hypothèses sont représentées par des traits pleins et la conclusion par un trait en pointillé.

Activité 4.3. Prouvez le lemme **step_bigstep**. Il s'agit d'une preuve par induction sur l'hypothèse **step** i env i' env'.

Finalement, on peut prouver le théorème final **star_bigstep** dont l'énoncé est donné plus haut, et dans le squelette.

Activité 4.4. Prouvez le théorème **star_bigstep**. Comme pour la preuve du lemme **star_seq**, on utilisera la tactique **dependent induction** sur l'hypothèse **star** i env Skip env'.

5 Un interpréteur pour le langage While

Les sémantiques que nous avons définies pour le moment sont des prédicats, *i.e.* des formules logiques, des propositions. Ce ne sont pas des sémantiques **exécutables**. On propose dans cette section d'écrire un interpréteur, *i.e.* une fonction qui calcule l'état final obtenu après l'exécution d'un programme (une instruction du langage) à partir d'un état initial donné.

On écrira les définitions et preuves de cette section dans le fichier **Interp.v**.

Notre but est donc d'écrire une fonction **exec_prog** qui prendra un environnement de départ **env**, une instruction **i** et qui retournera un environnement final, obtenu après l'exécution de **i** à partir de **env**. Cette fonction sera forcément récursive.

Comme vu en cours, Coq n'accepte que les fonctions récursives dont il peut prouver la terminaison. Malheureusement, on ne peut pas assurer la terminaison de notre fonction d'évaluation, par exemple pour le programme **while** (*true*) **do** **x := x + 1**.

Pour contourner ce problème, on propose d'utiliser un argument supplémentaire à notre fonction **exec_prog** : un entier naturel **fuel** qui représentera notre quantité de *carburant* restante, *i.e.* un nombre d'étapes de calcul qui décroît strictement à chaque appel récursif. Si, pendant l'exécution d'un programme, on se retrouve à cours de carburant, on arrêtera l'exécution du programme.

Pour formaliser la possibilité d'une erreur lors de l'exécution, on utilisera le type **option**. Voici la définition du type **option** (défini dans la librairie standard Coq) :

```

Inductive option (A : Type) : Type :=
  | Some (a: A)
  | None.

```

Ainsi, une valeur de type `option A` est soit `Some a`, où `a` est de type `A`, dénotant la présence d'une valeur de type `A`; soit `None`, dénotant l'absence de telle valeur. Le type `option` est très courant dans les langages de programmation fonctionnelle (OCaml, Haskell (le type s'appelle alors `Maybe`)), mais devient aussi courant dans d'autres langages de programmation (la classe `Optional<T>` depuis Java 8, la classe `optional<T>` dans la librairie boost pour C++, le type `Maybe<T>` depuis C++11, le type `Option<T>` en Rust...).

Ainsi, notre fonction devient :

```
Fixpoint exec_prog (fuel: nat) (env: state) (i: stmt) : option state := ...
```

Activité 5.1. Écrivez la fonction `exec_prog`. La commande `Compute` située sous la définition de la fonction devrait retourner `Some 55`. Il s'agit de l'exécution sur 30 pas de calcul d'un programme qui calcule la somme des entiers de 0 à 10.

On souhaite s'assurer que l'interprète est correct et complet :

- **Correction** : si l'interprète retourne un état, alors la sémantique à grands pas retourne le même état.

```
Theorem exec_prog_bigstep:
  forall fuel s i s',
    exec_prog fuel s i = Some s' ->
    bigstep s i s'.
```

- **Complétude** : si la sémantique à grands pas retourne un état, alors il existe une quantité de carburant telle que l'interprète retourne le même état :

```
Theorem bigstep_exec_prog:
  forall s i s',
    bigstep s i s' ->
    exists fuel,
      exec_prog fuel s i = Some s'.
```

Prouvons d'abord la correction de l'interprète.

Activité 5.2. Prouvez le théorème `exec_prog_bigstep`, qui établit la correction de l'interprète.

Pour prouver la complétude, il conviendra de prouver au préalable un lemme qui dit que si l'interprète donne un résultat avec une certaine quantité de carburant `f`, alors il donnera le même résultat avec plus de carburant :

```
Lemma exec_prog_more_fuel:
  forall f s i s',
    exec_prog f s i = Some s' ->
    forall f',
      f' >= f ->
      exec_prog f' s i = Some s'.
```

Activité 5.3. Prouvez le lemme `exec_prog_more_fuel`.

Activité 5.4. Prouvez la complétude de l'interprète, *i.e.* le théorème `bigstep_exec_prog`. Vous utiliserez la fonction `max : nat -> nat -> nat` qui donne le maximum de deux entiers naturels, et le lemme `exec_prog_more_fuel` que vous venez de prouver.

6 Logique de Hoare

Dans cette section, nous allons définir les règles de la logique de Hoare, et prouver qu'elles sont correctes vis-à-vis de la sémantique à grands pas. Tout cela se passe dans le fichier `Hoare.v`.

On définit le type `pred` des prédicats sur l'état comme `state -> Prop`. Ce sera le type des préconditions P et des postconditions Q dans un triplet de Hoare $\{ P \} i \{ Q \}$.

On définit ensuite ce qu'est un triplet de Hoare valide, via la définition `valid_hoare_triple` :

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

L'ensemble des règles de la logique de Hoare vues en cours est rappelée ci-dessous.

<p>SKIP</p> $\frac{}{\{ P \} \text{skip} \{ P \}}$	<p>CONSÉQUENCE</p> $\frac{P' \Rightarrow P \quad \{ P \} i \{ Q \} \quad Q \Rightarrow Q'}{\{ P' \} i \{ Q' \}}$
<p>SÉQUENCE</p> $\frac{\{ P \} i_1 \{ Q \} \quad \{ Q \} i_2 \{ R \}}{\{ P \} i_1 ; i_2 \{ R \}}$	<p>WHILE</p> $\frac{\{ P \wedge \llbracket e \rrbracket \} s \{ P \}}{\{ P \} \text{while} (e) \text{do} s \{ P \wedge \neg \llbracket e \rrbracket \}}$
<p>AFFECTATION</p> $\frac{}{\{ P[x \leftarrow E] \} x := E \{ P \}}$	<p>CONDITION</p> $\frac{\{ P \wedge \llbracket e \rrbracket \} i_1 \{ Q \} \quad \{ P \wedge \neg \llbracket e \rrbracket \} i_2 \{ Q \}}{\{ P \} \text{if} (e) \text{then} i_1 \text{else} i_2 \{ Q \}}$

Activité 6.1. Prouvez le théorème `hoare_skip`, qui correspond à la règle SKIP ci-dessus. Pour ce faire, vous utiliserez :

- la tactique `unfold` pour déplier la définition de `valid_hoare_triple`
- la tactique `inv` (définie dans `Tactics.v`) pour faire une analyse par cas sur des prédicats inductifs (*e.g.* `bigstep`)

Activité 6.2. Prouvez le théorème `hoare_seq`, qui correspond à la règle SÉQUENCE ci-dessus.

Activité 6.3. Prouvez le théorème `hoare_if`, qui correspond à la règle `CONDITION` ci-dessus.

Activité 6.4. Prouvez le théorème `hoare_assign`, qui correspond à la règle `AFFECTATION` ci-dessus.

Activité 6.5. Prouvez le théorème `hoare_strengthen_pre`, qui correspond à une partie de la règle `CONSÉQUENCE` ci-dessus.

Activité 6.6. Prouvez le théorème `hoare_weaken_post`, qui correspond à l'autre partie de la règle `CONSÉQUENCE` ci-dessus.

Activité 6.7. Prouvez le théorème `hoare_while`, qui correspond à la règle `WHILE` ci-dessus. On utilisera, comme précédemment, la tactique `dependent induction`.

Le théorème précédent est compliqué à utiliser : il faut que la postcondition ait exactement la forme `fun env => P env /\ ~ eval_condP env c`. On propose d'écrire une règle plus simple à utiliser :

```
Lemma hoare_while':
  forall (P Q : pred ) c s I,
    valid_hoare_triple (fun env => I env /\ eval_condP env c) s I ->
    (forall env, P env -> I env) ->
    (forall env, I env /\ ~eval_condP env c -> Q env) ->
    valid_hoare_triple P (While c I s) Q.
```

On remarque que l'on utilise l'invariant `I` qui vient de l'instruction elle-même `While c I s`.

Activité 6.8. Prouvez ce lemme.

Indice : pas besoin de déplier `valid_hoare_triple`, il suffit d'utiliser les règles précédemment prouvées.

En utilisant ce lemme, et l'ensemble des autres règles, on peut prouver la correction du programme qui calcule la factorielle, que l'on a étudié en cours. Le programme est défini comme suit :

```
Definition factorielle n :=
  Seq (Assign "res" (Const 1))
    (While (Lt (Const 0) (Var "n"))
      (fun env => env "res" * Zfact (env "n") = Zfact n)
      (Seq (Assign "res" (Mul (Var "res") (Var "n")))
        (Assign "n" (Sub (Var "n") (Const 1)))))).
```

Le programme est paramétré par la valeur n du nombre donné en entrée au programme, afin de pouvoir exprimer l'invariant. L'invariant est $(\text{fun env} \Rightarrow \text{env "res"} * \text{Zfact (env "n")} = \text{Zfact n})$, c'est-à-dire : au début et à la fin de chaque tour de boucle, la valeur de la variable **"res"** multiplié par la factorielle de la valeur de la variable **"n"** est égale à la factorielle de l'entier n .

Une première preuve est donnée dans le lemme `fact_correct_first_try`. Cette preuve est encore un petit peu trop compliquée : on doit appliquer les bonnes règles, trouver les bons prédicats sur l'environnement intermédiaires pour `hoare_seq`.

Nous allons nous intéresser à l'automatisation de cette preuve.

7 Plus faible précondition

On s'intéresse à calculer, pour une postcondition Q et une instruction i , la plus faible précondition P telle que $\{ P \} i \{ Q \}$. On note cette plus faible précondition $\text{wp}(i, Q)$. On rappelle la définition donnée en cours :

- $\text{wp}(x := e, Q) = \lambda\sigma. Q(\sigma[x \leftarrow e])$
- $\text{wp}(\text{skip}, Q) = \lambda\sigma. Q(\sigma)$
- $\text{wp}(i_1 ; i_2, Q) = \text{wp}(i_1, \text{wp}(i_2, Q))$
- $\text{wp}(\text{if } (c) \text{ then } i_1 \text{ else } i_2, Q) = \lambda\sigma. \begin{cases} \llbracket c \rrbracket_\sigma = \text{true} & \Rightarrow \text{wp}(i_1, Q)(\sigma) \\ \llbracket c \rrbracket_\sigma = \text{false} & \Rightarrow \text{wp}(i_2, Q)(\sigma) \end{cases}$
- $\text{wp}(\text{while } (c) \text{ invariant } \mathbb{I} \text{ do } i, Q) =$
 $\lambda\sigma. \mathbb{I}(\sigma) \wedge \begin{cases} \forall \sigma', \sigma \sim_{V(i)} \sigma' \Rightarrow \\ \text{pour tout état } \sigma' \text{ obtenu après quelques exécutions de } i \\ (\llbracket c \rrbracket_{\sigma'} = \text{true} \wedge \mathbb{I}(\sigma') \Rightarrow \text{wp}(i, \mathbb{I})(\sigma')) \\ \text{l'invariant est maintenu} \\ \wedge (\llbracket c \rrbracket_{\sigma'} = \text{false} \wedge \mathbb{I}(\sigma') \Rightarrow Q(\sigma')) \\ \text{et si la condition devient fausse, } Q \text{ est vrai} \end{cases}$

où :

- \mathbb{I} est un invariant de la boucle
- $\sigma \sim_S \sigma'$ signifie que les environnements sont d'accord pour toutes les variables exceptées celle dans l'ensemble S
- $V(i)$ est l'ensemble des variables affectées par l'instruction i

Cette définition vous est fournie dans le squelette (fonction `wp`). Prenez le temps de vous convaincre que la définition Coq correspond à ce que l'on a vu en cours et que vous comprenez bien chaque morceau de code. La définition de `wp` utilise une fonction auxiliaire `vars_affected` qui renvoie la liste des variables qui peuvent être modifiées par une instruction.

Le lemme `bigstep_vars_affect` montre que l'exécution d'une instruction s ne modifie que les variables qui sont affectées dans l'instruction s . Ce lemme sera nécessaire pour le théorème suivant.

Activité 7.1. Prouvez ce lemme. Vous aurez besoin du lemme `in_app_iff`, dont le type est le suivant :

```
in_app_iff:
  forall (A : Type) (l l' : list A) (a : A),
    In a (l ++ l') <-> In a l \ / In a l'
```

Avant de prouver le théorème principal, on propose de prouver le lemme `auto_hoare_while`, dont l'énoncé est donné dans le squelette. Il s'agit du lemme le plus technique de la preuve du théorème principal. Il montre que si la précondition de la boucle, telle que calculée par la fonction `wp`, éclatée en plusieurs hypothèses, est vraie d'un environnement `env1`, alors si on atteint un environnement `env2` avec la sémantique à grands pas depuis `env1` en exécutant `While c I s`, alors la postcondition `Q` est vérifiée. Les hypothèses `Itrue`, `CondTrue` et `CondFalse` représentent la plus faible précondition de la boucle.

Activité 7.2. Le début de la preuve de `auto_hoare_while` vous est donnée, pour vous simplifier la tâche. Complétez la preuve. Il vous faudra utiliser les hypothèses `CondTrue` et `CondFalse`, selon l'évaluation de la condition de boucle, ainsi que `IHs` pour montrer que si `wp(i, I)` est vraie d'un environnement `env1` et `bigstep env1 i env2`, alors `I` est vrai de `env2`.

Voici le théorème principal, qui assure que la précondition calculée par la fonction `wp s Q` est effectivement une précondition correcte, *i.e.* le triplet $\{ \varnothing s Q \} s \{ Q \}$ est valide.

Activité 7.3. Prouvez le théorème suivant :

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

Cette preuve se fait par induction sur `s`. La plupart des cas est simple, il suffit d'appliquer les règles que l'on a prouvées précédemment, ainsi que le lemme `auto_hoare_while` que l'on vient de prouver.

Attention, pour le cas de la boucle, il faudra déplier la définition de `valid_hoare_triple`.

Un dernier petit lemme pour simplifier l'utilisation de l'automatisation de la logique de Hoare : pour prouver $\{ P \} s \{ Q \}$, il suffit de prouver que pour tout environnement σ , $P(\sigma) \Rightarrow \text{wp}(s, Q)$.

Activité 7.4. Prouvez le lemme `auto_hoare'`.

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

On a enfin terminé de prouver les règles de la logique de Hoare ! On va pouvoir l'utiliser pour prouver la correction d'un certain nombre de programmes.

8 Prouver la correction de programmes

8.1 Échange

Le premier exemple que l'on propose est un programme qui échange deux variables `"x"` et `"y"`, en passant par une variable intermédiaire `"tmp"`.

Activité 8.1. Prouvez, en utilisant le lemme `auto_hoare`, que le programme `swap` est correct :

```
Lemma swap_correct:
  forall A B,
    valid_hoare_triple
      (fun env => env "x" = A /\ env "y" = B)
      swap
      (fun env => env "x" = B /\ env "y" = A).
```

Le début de la preuve vous est fourni. Vous pourrez utiliser un début similaire pour les autres preuves.

8.2 Affectation lente

Le second programme auquel on s'intéresse, `slow_assign`, est un exemple du cours. Il s'agit de la boucle :

```
while (x < b) invariant (x <= b) {
  x := x + 1;
}
```

La boucle est annotée avec l'invariant $x \leq b$, exprimé en Coq comme `(fun env => eval_expr env (Var "x") <= eval_expr env (Var "b"))`. Cet invariant permet la génération correcte de la précondition par la fonction `wp`.

Activité 8.2. Prouvez la correction de cet algorithme, *i.e.* si initialement `x` est plus petit ou égal à `b`, alors à la fin de l'exécution de l'algorithme, `x = b`.

```
Lemma slow_assign_correct:
  valid_hoare_triple
    (fun env => env "x" <= env "b")
    slow_assign
    (fun env => env "x" = env "b").
```

Encore une fois, le début de la preuve vous est donné : il ne reste plus qu'à manipuler les différents buts et hypothèses à votre disposition et à utiliser les tactiques classiques (`auto`, `split`, `lia`...).

8.3 Somme lente

On propose l'algorithme `dummy_sum`, pour faire la somme de deux entiers.

```
dummy_sum(X, Y) :=
  x := X;
  y := Y;
  while (0 < y) {
    x := x + 1;
    y := y - 1;
  }
```

Cet algorithme ne fonctionne que si Y est positif ou nul.

La définition `gcd` du squelette a comme invariant `(fun env => True)`. Bien que cette propriété soit effectivement un invariant de la boucle, il ne vous sera pas utile pour prouver la correction de l'algorithme.

Activité 8.3. Écrivez un invariant pour la boucle de `dummy_sum`. Votre invariant devra faire référence aux paramètres `x` et `y`, qui correspondent aux valeurs initiales des variables `"x"` et `"y"`.

Activité 8.4. Prouvez la correction de l'algorithme `dummy_sum`.

```
Theorem dummy_sum_correct x y:
  0 <= y ->
  valid_hoare_triple
    (fun _ => True)
    (dummy_sum x y)
    (fun env => env "x" = x + y).
```

8.4 PGCD

Le prochain algorithme, `gcd`, est l'algorithme d'Euclide pour calculer le plus grand diviseur commun de deux entiers :

```
euclide(x, y) :=
  while x != y {
    if x > y
    then x := x - y
    else y := y - x
  }
```

La définition `gcd` du squelette a comme invariant `(fun env => True)`. Bien que cette propriété soit effectivement un invariant de la boucle, il ne vous sera pas utile pour prouver la correction de l'algorithme.

Activité 8.5. Écrivez un invariant pour la boucle de `gcd`. Votre invariant devra faire référence aux paramètres `x` et `y`, qui correspondent aux valeurs initiales des variables `"x"` et `"y"`. Vous pourrez également vous inspirer du lemme `Z.gcd_sub_diag_r`, qui énonce une propriété sur le PGCD et la soustraction.

Activité 8.6. Prouvez la correction de cet algorithme :

```
Theorem gcd_correct:
  forall x y, valid_hoare_triple
    (fun env => env "x" = x /\ env "y" = y)
    (gcd x y)
    (fun env => env "x" = env "y" /\ Z.gcd x y = Z.abs (env "x")).
```

Vous utiliserez les lemmes suivants :

- `Z.gcd_sub_diag_r`
- `Z.gcd_comm`
- `Z.gcd_diag`

8.5 Factorielle

On retrouve le programme `factorielle` que l'on a étudié en cours. L'invariant est déjà écrit pour vous (c'est celui du cours).

Activité 8.7. Complétez la preuve de la correction de `factorielle`.

```
Theorem factorielle_correct:
  forall n, n >= 0 ->
    valid_hoare_triple
      (fun env => env "n" = n)
      (factorielle n)
      (fun env => env "res" = Zfact n).
```

Vous utiliserez les lemmes suivants :

- `Z.mul_1_l`
- `Z.mul_assoc`
- `Zfact_pos`
- `Zfact_neg`

8.6 Autres programmes et spécifications

Le reste du squelette contient d'autres algorithmes pour lesquels nous vous laissons trouver les invariants de boucle et prouver la correction. Pour chacun, nous vous donnons la spécification (l'énoncé du triplet de Hoare attendu), et une liste de commandes `Coq` qui vous indique les différents lemmes qui pourraient vous être utiles.