Exam Program Verification 2017/2018

12th Oct 2017, 11:00–12:45

Lecturer: Wishnu Prasetya

1. **Program Semantic** [1.5 pt].

   Consider a simple programming language L_0 where we can write a program like this:

   > **vars** `x`, `y`, `z` ;
   > **init** `x`> `0` && `10`>`y` ;
   > { `z`:=`0` ; **if** `x`/`y` > `2` **then** `z`:=`2` **else** `z`:=`1` }

   The **init**-part specifies allowed initial states: the program can only execute on an initial state on which the **init**-predicate would evaluate to true. If the program is invoked on such a state, it will then execute its body-statement. At the end, the program's final state will be returned. It will furthermore mark the state as either 'normal' or 'exceptional'. A state is marked as exceptional if the program terminates by throwing an exception. In the above case, this would happen if for example the value of `y` is 0 in the division `x/y`.

   The full syntax of our programming language is as follows:

Program	$\rightarrow$	**vars** *variables* ;	(variables declaration)
		init *expression* ;	(allowed initial state)
		body;	
variables	$\rightarrow$	one or more identifiers (variable-name) separated by ","	
body	$\rightarrow$	"{" one or more statements separated by ";" "}"	
statement	$\rightarrow$	*identifier* := *expression*	(assignment)
	|	**if** *expression* **then** *body* **else** *body*	
expression	$\rightarrow$	integer constants like 0,1,2,...	
	|	"`undefined`"	
	|	*identifier*	
	|	*expression* + *expression*	
	|	*expression* / *expression*	(the div operator)
	|	*expression* == *expression*	(testing equality)
	|	*expression* > *expression*	(greater than)
	|	*expression* && *expression*	('and' operator)

   - e_1/e_2 results in `undefined` if e_2 evaluates to zero.
   - All the binary operators above, e_1 *op* e_2, result in `undefined` if one of its arguments evaluates to `undefined`.
   - An assignment x:=e throws an exception if e evaluates to `undefined`. The program will then break its execution, and its state is left as it was just before the assignment.
   - The statement **if** e **then** ... **else** ... throws an exception if e evaluates to `undefined`. The program will then break its execution, and its state is left as it was just before the **if**.

Your tasks:

(a) *Provide a denotational semantic* for the above programming language. You will need to provide the functions $\mathcal{E}$, $\mathcal{S}$, and $\mathcal{P}$ that describe the semantic of respectively expressions, statements, and programs. Hint: choose a proper semantical domain for each.

(b) Suppose we extend the syntax so that we can also write a post-condition. A post-condition will be written as a pair **post** p **exceptional** q to mean that the program should either terminate normally in a state satisfying p, or terminate exceptionally in a state satisfying q. For example, the post-condition below is a valid one:

> **vars** `x`, `y`, `z` ;
> **init** `x`> `0` && `10`>`y` ;
> **post** `z`> `0` **exceptional** `z==0`
> { `z:=0` ; **if** `x/y` > `2` **then** `z:=2` **else** `z:=1` }

Extend your denotational semantic so that the above semantic of post-condition is defined (and reasonable).

Answer:

Grading: a=1, b=0.5.

Let Val be a domain of values consisting of integers and boolean values. Let $\perp$ be a shorthand for `undefined`. Let $Val^\perp$ be $Val \cup \{\perp\}$.

Let $Vars$ be the universe of variable names. We will represent a state with a function of this type: $State = Vars \to Val$. Because S_0 only have global variables, in the following semantic we will not make explicit which variables are actually in the scope.

The semantic of expressions is defined by $\mathcal{E} : expr \to State \to Val^\perp$ as follows:

(a) The definition for literals is obvious, with this addition: $\mathcal{E}[\texttt{undefined}] = (\lambda s.\ \perp)$.

(b) $\mathcal{E}[x] = (\lambda s.\ s\ x)$.

(c) $\mathcal{E}[e_1/e_2] = (\lambda s.\ \ \textbf{if}\ v_1{=}\perp \vee\, v_2{=}\perp \vee\, v_2{=}0\ \textbf{then}\ \perp\ \textbf{else}\ v_1/v_2\ )$
$\qquad\qquad\qquad\textbf{where}$
$\qquad\qquad\qquad v_1 = \mathcal{E}[e_1]\ s\ ,\ v_2 = \mathcal{E}[e_2]\ s$

(d) For other binary operators $\oplus$:
$\mathcal{E}[e_1\text{"}\oplus\text{"}e_2] = (\lambda s.\ \ \textbf{if}\ v_1{=}\perp \vee\, v_2{=}\perp\ \textbf{then}\ \perp\ \textbf{else}\ v_1 \oplus v_2\ )$
$\qquad\qquad\qquad\textbf{where}$
$\qquad\qquad\qquad v_1 = \mathcal{E}[e_1]\ s\ ,\ v_2 = \mathcal{E}[e_2]\ s$

We will flag states with either N or E to mark it as either normal or exceptional. Let $State^+ = State \times \{\mathsf{N}, \mathsf{E}\}$; so it is the domain of marked states.

The semantic of statements is defined by $\mathcal{S} : State^+ \to State^+$ as follows:

(a) A body is either empty of a non-empty list of statements:

$\mathcal{S}[[]]\ (s, m)$	$=$	(s, m)	– empty body does nothing
$\mathcal{S}[S_1 : rest]\ (s, \mathsf{E})$	$=$	(s, m)	– if we are already in an exceptional state
$\mathcal{S}[S_1 : rest]\ (s, \mathsf{N})$	$=$	$\mathcal{S}[rest]\ (\mathcal{S}[S_1]\ (s, \mathsf{N}))$	

(b) Assignment, we only need to define it when it is executed on a normal state:
$\mathcal{S}[v := e]\ (s, \mathsf{N})\ =\ \ \textbf{if}\ v{=}\perp\ \textbf{then}\ (s, \mathsf{E})\ \textbf{else}\ (update\ s\ x\ v, \mathsf{N})$
$\qquad\qquad\qquad\textbf{where}$
$\qquad\qquad\qquad v = \mathcal{E}[e]\ s$

(c) If-then-else, we only need to define it when it is executed on a normal state:

$$\begin{aligned}\mathcal{S}[\textbf{if } g \textbf{ then } S_1 \textbf{ else } S_2]\ (s,\mathsf{N}) \;=\;& \textbf{if } v{=}\bot \textbf{ then } (s,\mathsf{E}) \textbf{ else } (\textbf{if } v{=}\mathit{true} \textbf{ then } r_1 \textbf{ else } r_2)\\ & \textbf{where}\\ & v = \mathcal{E}[g]\ s\\ & r_1 = \mathcal{S}[S_1]\ (s,\mathsf{N})\\ & r_2 = \mathcal{S}[S_2]\ (s,\mathsf{N})\end{aligned}$$

The semantic of a whole program is defined as a function from states that satisfy the `init`-predicate to State^+. For simplicity, states that do not satisfy the `init`-predicate will be mapped to $\bot$. This is defined through the semantic function $\mathcal{P} : \mathit{Program} \to \mathit{State} \to (\mathit{State}^+ \cup \{\bot\}$ as follows:

$$\mathcal{P}[\texttt{vars...}\ \texttt{init}\ p\ \mathit{body}]\ s \;=\; \textbf{if } \mathcal{E}[p]\ s = \mathit{true} \textbf{ then } \mathcal{S}[\mathit{body}]\ (s,\mathsf{N}) \textbf{ else } \bot$$

For the second question, where a program is extended with a post-condition to form a specification. The specification is valid if the program always ends with the specified post-condition. We define the semantic function: $\mathcal{H}\mathit{oare} : \mathit{Spec} \to \mathit{Bool}$ as follows:

$$\begin{aligned}&\mathcal{H}\mathit{oare}[\texttt{vars...}\ \texttt{init}\ p\ \texttt{post}\ q_N\ \texttt{exceptional}\ q_E\ \ \texttt{body}]\ s\\ &=\\ &(\forall s.\quad \textbf{case } \mathcal{P}[\texttt{vars...}\ \texttt{init}\ p\ \mathit{body}]\ s \textbf{ of }\ )\\ &\qquad (t,\mathsf{N}) \to (\mathcal{E}[q_N]\ t = \mathit{true})\\ &\qquad (t,\mathsf{E}) \to (\mathcal{E}[q_E]\ t = \mathit{true})\\ &\qquad \bot \to \mathit{true}\end{aligned}$$

2. **Loop Invariant** [1.5 pt].

Give an invariant for each of the GCL loops below. It should be an invariant that is consistent, strong enough to realize the asked post-condition, and realistic to be established by the pre-condition or initialization of the loop. Use the *partial correctness* interpretation of Hoare triples.

Below, `a` is an infinite array of `int`; `b` is of type `bool`; other variables are of type `int`.

(a) $\{\ \texttt{x} = \texttt{100}\ \}$ **while** $\texttt{x}{>}\texttt{0}$ **do** $\{\texttt{x} := \texttt{x}{-}\texttt{2}\ \}$ $\{\ \texttt{x}{=}\texttt{0}\ \}$

Answer: $x \geq 0 \;\wedge\; \mathit{even}(x)$

(b) $\{\ \texttt{x}{=}\texttt{10} \wedge \texttt{y}{=}\texttt{0}\ \}$ **while** $\texttt{x}{>}\texttt{0}$ **do** $\{\texttt{x} := \texttt{x}{-}\texttt{1}\ ;\ \texttt{y} := \texttt{y}{+}\texttt{10}\ \}$ $\{\ \texttt{x}{+}\texttt{y}{=}\texttt{100}\ \}$

Answer: $x{\geq}0 \wedge 10x{+}y{=}100$

(c) $\{\ \texttt{x}{=}\texttt{100} \wedge \texttt{y}{=}\texttt{1}\ \}$ **while** $\texttt{x}{>}\texttt{y}$ **do** $\{\texttt{y} := \texttt{y}{*}\texttt{2}\ \}$ $\{\ \texttt{y}{=}\texttt{128}\ \}$

Answer: $(\exists k : k{\geq}0 : y = 2^k) \;\wedge\; y{\leq}128 \;\wedge\; x{=}100$. The last conjunct can be kept implicitly since the program does not modify x.

(d) Here is a program to check if an array consists of only 0's:

$\{\ \texttt{k}{=}\texttt{0} \;\wedge\; \texttt{allzeros}{=}\textbf{true}\ \}$

while $\texttt{k}{<}\texttt{N} \wedge \texttt{allzeros}$ **do** $\{\ \texttt{allzeros} := (\texttt{a[k]}{=}\texttt{0})\ ;\ \texttt{k}{:=}\texttt{k}{+}\texttt{1}\ \}$

$\{\ \texttt{allzeros} \;=\; (\forall \texttt{i} : \texttt{0}{\leq}\texttt{i}{<}\texttt{N} : \texttt{a[i]} = \texttt{0})\ \}$

Answer: $allzeros = (\forall i : 0 \leq i < k : a[i]{=}0)$
$\wedge$
$0{\leq}k \wedge (k{\leq}N \vee N < 0)$

The $N < 0$ part is for the case that the program starts with negative N, in which case it will immediately break the loop. I will not count it wrong if you forget this part.

(e) $\{$ **true** $\}$

`k, found` $:=$ `0,` **false** ;
while $\neg$`found` **do** $\{$ `found` $:=$ `(a[k]=0)` ; `k:=k+1` $\}$

$\{\ \mathtt{k} \geq \mathtt{0}\ \wedge\ \mathtt{a[k{-}1]{=}0}\ \}$

Answer: $found = (\exists i : 0 \leq i < k : a[i] = 0)$
$\wedge\ \ (\forall i : 0 \leq i < k{-}1 : a[i] \neq 0)$
$\wedge\ \ 0{\leq}k$

3. **Weakest pre-condition** [1.5 pt].

(a) Consider the loop below, with the given post-condition; x is of type integer:

$$\textbf{while } \texttt{x>0} \textbf{ do } \{ \textbf{ assert } \texttt{even(x)} \ ; \ \texttt{x := x-2} \ \} \ \ \{ \ \texttt{even(x)} \ \}$$

where $even(x)$ is a predicate that means that x is an even integer.

Calculate the wlp of the loop above using the fix-point iteration.

Answer:

$W_0 = true$

$W_{i+1} \ = \ (x \leq 0 \wedge even(x)) \ \vee \ (x > 0 \wedge \mathsf{wlp}\ body\ W_i)$

So, we get as W_1:

$W_1 \ = \ (x \leq 0 \wedge even(x)) \ \vee \ (x > 0 \wedge even(x)$, which can be simplified to simply $even(x)$.

Then W_2:

$W_2 \ = \ (x \leq 0 \wedge even(x)) \ \vee \ (x > 0 \wedge even(x) \wedge even(x-2)$, which again is equivalent to $even(x)$.

Hence we reach the a fix point, thus concluding that the wlp is $even(x)$.

(b) Suppose we want to have a non-deterministic conditional statement in our programming language. We will denote it with the following multi-armed **if**, with $n \geq 1$:

$$\begin{array}{lll} \textbf{if}\ g_1 & \rightarrow & S_1 \\ \quad ... & & \\ \quad g_n & \rightarrow & S_n \end{array}$$

$g_1 ... g_n$ are 'guards'; these are boolean expressions. $S_1 ... S_n$ are statements.

This is how the above statement works. Suppose we execute it on a state s. If there are multiple guards that evaluate to true on s, one will be selected non-deterministically, e.g. g_k, and the corresponding S_k is then executed.

If no guard evaluates to true on s, the whole statement simply does a skip.

Give a reasonable definition of the wlp of such a statement.

Answer: The wlp wrt to a post-condition Q is:

$$(\forall k : 1 \leq k \leq n : \ g_k \Rightarrow \mathsf{wlp}\ S_k\ Q) \ \wedge \ ((\forall k : 1 \leq k \leq n : \ \neg g_k) \Rightarrow Q)$$

Note that converting this to a disjunctive form as we did to a normal if-then-else does not give an equivalent formula (and wrong):

$$(\exists k : 1 \leq k \leq n : \ g_k \wedge \mathsf{wlp}\ S_k\ Q) \ \vee \ ((\forall k : 1 \leq k \leq n : \ \neg g_k) \wedge Q)$$

It worked with the normal if-then-else because the guard can only be either g or $\neg g$; so there is no non-determinism there.

(c) Give the definition of **repby** and propose a definition of the wlp of assignments that target a two dimensional array.

Answer:

$$a(i, j\ \mathbf{repby_2}\ e) \ = \ a(i\ \mathbf{repby_1}\ (a[i](j\ \mathbf{repby_1}\ e)))$$

So, $a(i, j\ \mathbf{repby_2}\ e)[i][j] \ = \ a(i\ \mathbf{repby_1}\ (a[i](j\ \mathbf{repby_1}\ e)))[i][j]$, which is equal to:

$$(a[i](j\ \mathbf{repby_1}\ e))[j]$$

which is equal to e. If we assume $j \neq 0$, then notice that $a(i, j\ \mathbf{repby_2}\ e)[i][0]$ by applying the same unfolding is:

$$(a[i](j\ \mathbf{repby_1}\ e))[0]$$

which is then equal to $a[i][0]$.

4. **Basic HOL** [1 pt].

   (a) In HOL, a tactic is a function of the type:

   $$goal \to (goal\ \texttt{list}\ \#\ proofFunction)$$

   where $goal = (\texttt{term list}\ \#\ \texttt{term})$ and $proofFunction = \texttt{thm list} \to \texttt{thm}$.
   The combinator `THEN` : $tactic \to tactic \to tactic$ applies two tactics one after another. That is, t_1 `THEN` t_2 applies t_1 on the given goal, then it applies t_2 on all the subgoals produced by t_1. Note that `THEN` produces a new tactic (you can see it in its type!) that internally does what is said in the previous sentence.
   Give the definition of `THEN`. You can give the definition in terms of a pseudo-code (it does not have to be in ML).
   Answer:
   $(t_1\ \texttt{THEN}\ t_2) g$
   $=$
   let
   $(subgoals, pf_1) = t_1\ g$
   $(moregoals, pfs)\ = unzip\ (map\ t_2\ subgoals)$
   $lengths\ =\ map\ length\ moregoals$
   in
   $(concat\ moregoals, (\lambda thmlist.\ pf_1\ [f\ thms \mid (f, thms) \in pfs \times segmentize\ lengths\ thmlist]))$
   where $segmentize\ ns\ s$ divides s in segments of lengths as in ns:
   $segmentize\ []\ []\ =\ []$
   $segmentize\ (n : rest)\ s\ =\ take\ n\ s\ :\ segmentize\ rest\ (drop\ n\ s)$

   (b) Show how the quantifiers $\forall$ and $\exists$ are defined in the primitive HOL. If you use operators other than function application, λ, $=$, $\Rightarrow$, and `T` define your operators as well.
   Answer:

   $$\forall P\ =\ (P = (\lambda x.\ T))$$
   $$\exists P\ =\ P(@P)$$

   where @ is defined through an axiom, namely, for all P, x, $P\ x \Rightarrow P(@P)$

5. **Hoare Logic** [0.5 pt, challenging].

   Consider again the language L_0 in the question No. 1. Propose how to calculate the wlp of the statements in L_0. Keep in mind that we have defined a post-condition in L_0 to be a pair of predicates Q_N, Q_E specifying the program final state when it ends normally, and when it ends exceptionally.

   Answer: The idea is to define $\mathsf{wlp}\ S\ Q_N$ to calculate the weakest pre-condition (a single predicate) so that we will end up either normally in Q_N or exceptionally in Q_E. Below Q_E is always the same Q_E as the top-level Q_E:

   (a) Assignment. We take into account that the program break if e evaluates to $\perp$, in which case the resulting state should satisfy Q_E.
   $\mathsf{wlp}\ (x := e)\ Q_N\ =\ (e \neq \perp \Rightarrow Q_N[e/v]) \wedge (e = \perp \Rightarrow Q_E)$

   (b) If-then-else. We take into account that the program break if g evaluates to $\perp$, in which case the resulting state should satisfy Q_E.
   $\mathsf{wlp}\ (\textbf{if}\ g\ \textbf{then}\ S_1\ \textbf{else}\ S_e)\ Q_N\ =\ (g \neq \perp \wedge g \Rightarrow \mathsf{wlp}\ S_1\ Q_N) \wedge (g \neq \perp \wedge \neg g \Rightarrow \mathsf{wlp}\ S_2\ Q_N) \wedge (g = \perp \Rightarrow Q_E)$

   (c) The wlp of sequential composition (in a body) can be defined as usual:
   $\mathsf{wlp}\ []\ Q_N\ =\ Q_N$
   $\mathsf{wlp}\ (S_1; rest)\ Q_N\ =\ \mathsf{wlp}\ S_1\ (\mathsf{wlp}\ rest\ Q_N)$

6. **HOL** [4 subquestions for total 4 pt, time: 48 hrs].

From the PV website, you can download the file hol_exam1718.smx. This is basically the same as in the HOL-tutorial.

It contains the following parts:

Section 1 defines an embedding of a subset of GCL in HOL. It also contains an example of how a simple GCL program is expressed in HOL.

Section 2 defines the semantic of GCL constructs, the semantic of Hoare triple, and provides a definition of wlp.

Section 3 provides the proofs of some basic laws of Hoare logic, for example these:

pre-condition strengthening:

$$\frac{\{q\}\ \mathit{stmt}\ \{r\} \quad , \quad p \Rightarrow q}{\{p\}\ \mathit{stmt}\ \{r\}}$$

post-condition weakening:

$$\frac{\{p\}\ \mathit{stmt}\ \{q\} \quad , \quad q \Rightarrow r}{\{p\}\ \mathit{stmt}\ \{r\}}$$

Section 4 proves the soundness the wlp defined in Section 3. 'Sound' here means that any final state that results from executing a GCL statement *stmt* from any state in the pre-condition produced by wlp *stmt* q will satisfy q. In other words, the following Hoare triple is always valid:

$$\{ \ \mathsf{wlp}\ \mathit{stmt}\ q \ \} \quad \mathit{stmt} \quad \{ \ q \ \}$$

for any GCL statement *stmt*.

Section 5 shows how to prove the correctness of the example from Section 1, with respect to some post-condition.

The problems that you have to solve are listed below. Send your solution in the form of a modified script hol_exam_1718.smx, that contains all your proofs. Rename the file to yourname_hol_exam_1718.smx, and mark the parts that contain your proofs.

(a) (0.5 pt) Extend Section 5. Give a representation of a GCL statement that calculates $min(x, y)$ and stores the result in x. **Prove** that your statement satisfies this Hoare triple:

$$\{(x = X) \land (y = Y)\} \text{ your-stmt } \{((x = X) \lor (x = Y)) \land x \leq X \land x \leq Y\}$$

Do note that in HOL, "=" has a low priority; so you usually need to bracket its use.

Also note that to finish (directly prove) a goal that purely involves integer arithmetic, you can use `COOPER_TAC` (rather than `PROVE_TAC`; in fact, the latter won't work in such a situation).

(b) (0.5 pt) Extend Section 3. Prove the following law about Hoare triples (I will spell out the law this time):

For any statement *stmt* (in our GCL), and any predicates p_1, p_2, q_1, q_2: if $\{p_1\}$ *stmt* $\{q_1\}$ and $\{p_2\}$ *stmt* $\{q_2\}$ are two valid specifications, then the following are also valid:

i. $\{p_1 \land p_2\}\ \mathit{stmt}\ \{q_1 \land q_2\}$

ii. $\{p_1 \lor p_2\}\ \mathit{stmt}\ \{q_1 \lor q_2\}$

(c) (1 pt) Introduce a concept of program refinement. A GCL statement $stmt_1$ is said to *refine* another statement $stmt_2$ if all Hoare triple specifications that are valid for $stmt_2$ are also valid for $stmt_1$.

Task: propose a reasonable condition under which the GCL statement "`assume` p ; $stmt_1$" would **refine** the statement "`assume` p ; `if` g `then` $stmt_1$ `else` $stmt_2$".

Prove your claim in HOL (put it in Section-3).

(d) (1 pt) You need to extend Section 1 and Section 2. We want to introduce a new construct:

PERM $stmt_1$ $stmt_2$

This will execute either the sequence $stmt_1$; $stmt_2$ or the sequence $stmt_2$; $stmt_1$. The choice is non-deterministic.

Propose the wlp of such a construct, and **prove the soundness of your proposal**. That is, you want to prove:

$$\{\ \mathsf{wlp}\ (\mathsf{PERM}\ stmt_1\ stmt_2)\ q\ \}\ \ \mathsf{PERM}\ stmt_1\ stmt_2\ \ \{\ q\ \}$$

Hint: add a non-deterministic-choice operator to GCL, then re-prove `SOUND_wlp_thm`.

(e) (0.5pt) Consider the following construct of `for`-loop:

`for` *invariant* (*init* ; g ; *incr*) *body*

where *init* is an assignment; g is a predicate; *incr* and *body* are statements; and *invariant* is a proposed invariant for the loop.

The execution of such a loop proceeds as 'usual', namely as follows. First *init* is executed, then we start to iterate. If the guard g evaluates to true we will do the *body* followed by *incr*. Then g is evaluated again. If it is true, we do another iteration and so on. If g is false when it is evaluated, the loop terminates.

Prove that the wlp of such a loop is sound. This should be quite easy if you have done the H3 assignment.

(f) (0.5, challenging) Extend Section 3. Someone proposes the following law on Hoare triples:

$$\frac{\{p_1\}\ stmt\ \{q_1\}\quad ,\quad \{p_2\}\ stmt\ \{q_2\}}{\{p_1 \vee p_2\}\ stmt\ \{q_1 \wedge q_2\}}$$

Please notice the subtle difference with the previous law in 6b. **Prove** or **disapprove** this law. (yes, you can disapprove a law proposal in HOL, because it is higher order, but you would first need to formulate the disapproval).