Technische Universität München
Fakultät für Informatik
Dr. Peter Lammich
Simon Wimmer

WS 2018/19
20.11.2018

Semantics of Programming Languages

Exercise Sheet 6

Exercise 6.1 Weakest Preconditions

In this exercise, you shall prove the correctness of two simple programs using weakest preconditions.

Step 1 Write a program that stores the maximum of the values of variables a and b in variable c.

definition *Max* :: *com* **where**

Step 2 Prove these lemmas about *max*:

lemma [*simp*]: "$(a{::}int){<}b \Longrightarrow max\ a\ b = b$"

lemma [*simp*]: "$\neg(a{::}int){<}b \Longrightarrow max\ a\ b = a$"

Show total correctness of *Max*:

lemma "$wp\ Max\ (\lambda s.\ s\ ''c'' = max\ (s\ ''a'')\ (s\ ''b''))\ s$"

Step 3 Note that our specification still has a problem, as programs are allowed to overwrite arbitrary variables.
For example, regard the following (wrong) implementation of *Max*:

definition "$MAX\_wrong = (''a''{::=}N\ 0;;''b''{::=}N\ 0;;''c''{::=}\ N\ 0)$"

Prove that *MAX_wrong* also satisfies the specification for *Max*:

lemma "$wp\ MAX\_wrong\ (\lambda s.\ s\ ''c'' = max\ (s\ ''a'')\ (s\ ''b''))\ s$"

What we really want to specify is, that *Max* computes the maximum of the values of a and b in the initial state. Moreover, we may require that a and b are not changed.

For this, we can use logical variables in the specification. Prove the following more accurate specification for *Max*:

lemma "$a{=}s\ ''a'' \wedge b{=}s\ ''b'' \Longrightarrow wp\ Max\ (\lambda s.\ s\ ''c'' = max\ a\ b \wedge a = s\ ''a'' \wedge b = s\ ''b'')\ s$"

Step 4 Write a program that calculates the sum of the first n natural numbers. The parameter n is given in the variable n.

definition *Sum* :: *com* **where**

Step 5 Find a proposition that states *partial* correctness of *Sum* and prove it! *Hint:* Use the following specification for the sum of the first n non-negative integers.

fun *sum* :: *"int ⇒ int"* **where**
"sum i = (if i ≤ 0 then 0 else sum (i − 1) + i)"

lemma *sum_simps*[*simp*]:
  "0 < i ⟹ sum i = sum (i − 1) + i"
  "i ≤ 0 ⟹ sum i = 0"
  by *simp+*

lemmas [*simp del*] = *sum.simps*

Exercise 6.2 Forward Assignment Rule

Think up and prove a forward assignment rule, i.e., a rule of the form $P\ s \Longrightarrow wp\ (x ::= a)\ Q\ s$, where Q is some suitable postcondition.

lemmas *fwd_Assign′* = *wp_conseq*[*OF fwd_Assign*]

Redo the proofs for *Max* from the previous exercise, this time using your forward assignment rule.

lemma *"wp Max (λs. s ''c'' = max (s ''a'') (s ''b'')) s"*

Homework 6.1 Squaring

Submission until Tuesday, November 27, 2018, 10:00am.

Consider the following program for squaring a non-negative integer:

definition *"square ≡*
  ''z'' ::= N 1;;
  ''a'' ::= N 0;;
   WHILE Less (N 0) (V ''n'') DO (
    ''a'' ::= Plus (V ''a'') (V ''z'');;
    ''z'' ::= Plus (V ''z'') (N 2);;
    ''n'' ::= Plus (V ''n'') (N (−1))
)"

Here is a picture illustrating the algorithm:

```
1 2 3 4
2 2 3 4
3 3 3 4
4 4 4 4
```

Prove that this algorithm correctly squares non-negative integers:

theorem *"s ''n'' ≡ n ⟹ n ≥ 0 ⟹ wlp square (λs'. let a = s' ''a'' in a = n∗n) s"*

Hints:

- Your invariant should state something about all three variables, n, a, and z.
- The simp rules from *algebra_simps* can help to solve any remaining verification conditions.

Homework 6.2 IMP Interpreter

Submission until Tuesday, November 27, 2018, 10:00am.

The goal of this exercise is to define an interpreter for IMP programs and to prove it correct. First define a function *cfg_step* that interprets a given configuration for a single step:

fun *cfg_step* :: *"com ∗ state ⇒ com ∗ state"* **where**

Your function should fulfill the following properties:

theorem *small_step_cfg_step*: *"cs → cs' ⟹ cfg_step cs = cs'"*

theorem *final_cfg_step*: *"final cs ⟹ cfg_step cs = cs"*

Prove these properties!

Our interpreter will interpret programs with a finite amount of fuel, i.e. it simply iterates *cfg_step* for a finite number of times:

fun *cfg_steps* :: *"nat ⇒ com ∗ state ⇒ com ∗ state"* **where**
  "cfg_steps 0 cs = cs" |
  "cfg_steps (Suc n) cs = cfg_steps n (cfg_step cs)"

Prove that the interpreter is complete:

theorem *small_steps_cfg_steps*:
  "cs →∗ cs' ⟹ ∃n. cfg_steps n cs = cs'"

and that it is sound:

theorem *cfg_steps_small_steps*:
  "$cfg\_steps\ n\ cs = cs' \Longrightarrow cs \rightarrow\!* \ cs'$"

corollary *cfg_steps_correct*:
  "$cs \rightarrow\!* \ cs' \longleftrightarrow (\exists\, n.\ cfg\_steps\ n\ cs = cs')$"
  by (*metis small_steps_cfg_steps cfg_steps_small_steps*)

Homework 6.3 Simulation and Termination

Submission until Tuesday, November 27, 2018, 10:00am.

Note: This is a bonus exercise worth up to three bonus points.

In this exercise, we consider an abstract notion of simulation between transition system. We simply model transition systems as relations of type $'a \Rightarrow 'a \Rightarrow bool$. A system $step'$ simulates a systems *step* with respect to relation R if the following holds:

definition
  "$is\_sim\ R\ step\ step' \equiv \forall\, a\ b\ a'.\ R\ a\ b \wedge step\ a\ a' \longrightarrow (\exists\, b'.\ R\ a'\ b' \wedge step'\ b\ b')$"

First show that this simulation property can also be extended to runs in the transition system:

lemma *is_sim_star*:
  assumes "$is\_sim\ R\ step\ step'$" "$R\ a\ b$" "$step^{**}\ a\ a'$"
  shows "$\exists\, b'.\ R\ a'\ b' \wedge step'^{**}\ b\ b'$"

Define an inductive predicate that correctly characterizes the notion of a *terminating* state. The predicate *terminating step s* should hold if there is no infinite execution path from s in the system specified by *step*:

inductive *terminating* **for** *step* **where**

Prove the following theorem that connects simulation and termination:

theorem *terminating_simulation*:
  assumes "$is\_sim\ R\ step\ step'$" "$terminating\ step'\ b$" "$R\ a\ b$"
  shows "$terminating\ step\ a$"

Does the converse also hold?

Homework 6.4 Challenge: Partial Correctness of While

Submission until Tuesday, November 27, 2018, 10:00am.

This is a bonus exercise. The challenge is to find a proof of the theorem *wlp_whileI'* that is as short as possible. The shortest solution gets three points. Solutions that get close to it will get partial credit. The length of a solution is the number of tokens it contains. We will count tokens of the outer syntax roughly as follows:

- Terms of the inner syntax such as *abc*, $a + b$ etc. will all be counted as one token.
- Keywords and keyword tokens of the outer syntax such as *apply*, *done*, *by*, *proof*, *next*, (,), [,], *induction*, *auto*, :, *add*, *of*, *where*, *OF* etc. will all be counted as one token.
- Whitespace is not a token
- You are not allowed to just use the existing fact wlp_whileI'