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*:
"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.

lemma
fwd_Assign: *"P s ⟹ wp (x::=a) (λs′. ∃s. P s ∧ s′=s(x := aval a s)) s"*

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"*

Exercise 6.3 Weakest Preconditions for *OR*

Use the extend version of IMP with c_1 *OR* c_2. Recall that it models nondeterministic choice: it may execute either c_1 or c_2. Add a rule for *OR* to the weakest precondition calculus in theory *Wp_Demo*, and adjust the proofs.

Homework 6.1 While Invariants

Submission until Tuesday, November 20, 2018, 10:00am.

We have pre-defined a while-combinator

$$while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$$

such that the following unfolding property holds:

$$while\ b\ f\ s = (if\ b\ s\ then\ while\ b\ f\ (f\ s)\ else\ Some\ s)$$

To prove anything about the computation result of *while* we need to use a proof rule with an invariant (similarly to what you have seen for the weakest precondition calculus). Prove that the following rule is correct:

theorem *while_invariant*:
  assumes "*wf R*" **and** "*I s*"
    and "$\bigwedge s.\ I\ s \Longrightarrow b\ s \Longrightarrow I\ (f\ s) \wedge (f\ s,\ s) \in R$"
  shows "$\exists s'.\ I\ s' \wedge \neg\ b\ s' \wedge while\ b\ f\ s = Some\ s'$"
  using *assms*(*1*,*2*)
proof *induction*
  case (*less s*)

Here is an example of how we can use this rule:

definition
  "$list\_sum\ xs \equiv fst\ (the\ (while\ (\lambda(s,\ xs).\ xs \neq [])\ (\lambda(s,\ xs).\ (s + hd\ xs,\ tl\ xs))\ (0,\ xs)))$"

lemma *list_sum_list_sum*:
  "*list_sum xs = sum_list xs*"
proof −
  let *?I* =
  let $?R$ = "$\{((s,\ as),\ (s',\ bs)).\ length\ as < length\ bs \wedge length\ bs \leq length\ xs\}$"
  have "*wf ?R*"
    by (*rule wf_bounded_measure*[**where**
        ub = "$\lambda\_.\ length\ xs$" **and** f = "$\lambda(\_,\ ys).\ length\ xs - length\ ys$"]) *auto*
  have "$\exists\ s'.\ ?I\ s' \wedge \neg\ (\lambda(s,\ xs).\ xs \neq [])\ s' \wedge$
    $while\ (\lambda(s,\ xs).\ xs \neq [])\ (\lambda(s,\ xs).\ (s + hd\ xs,\ tl\ xs))\ (0,\ xs) = Some\ s'$"
    apply (*rule while_invariant*[*OF* ‹*wf ?R*›])
     apply *simp*
    apply *clarsimp*
    subgoal for *zs ys*
      apply (*rule exI*[**where** x = "*ys* @ [*hd zs*]"])
      apply *auto*
      done
    done
  then show *?thesis*
    unfolding *list_sum_def* **by** *auto*
qed

You can get one bonus point if you manage to fill in an invariant for *?I* such that the proof goes through!

Homework 6.2 IMP Interpreter

Submission until Tuesday, November 20, 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′ ⟹ cs →∗ cs′"

corollary *cfg_steps_correct*:
  "cs →∗ cs′ ⟷ (∃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 20, 2018, 10:00am.

Note: This is a bonus exercise worth three additional points.

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

definition

$$\text{“}\mathit{is\_sim}\ R\ \mathit{step}\ \mathit{step}' \equiv \forall\, a\ b\ a'.\ R\ a\ b \wedge \mathit{step}\ a\ a' \longrightarrow (\exists\, b'.\ R\ a'\ b' \wedge \mathit{step}'\ b\ b')\text{”}$$

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

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

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 “$\mathit{is\_sim}\ R\ \mathit{step}\ \mathit{step}'$” “$\mathit{terminating}\ \mathit{step}'\ b$” “$R\ a\ b$”
  shows “$\mathit{terminating}\ \mathit{step}\ a$”

Does the converse also hold?

Homework 6.4 Challenge: Partial Correctness of While

Submission until Tuesday, November 20, 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