Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

WS 2022/23
22.11.2022

Semantics of Programming Languages

Exercise Sheet 5

Exercise 5.1 Program Equivalence

Let *Or* be the disjunction of two *bexp*s:

definition *Or* :: *"bexp ⇒ bexp ⇒ bexp"* **where**
"Or b1 b2 = Not (And (Not b1) (Not b2))"

Prove or disprove (by giving counterexamples) the following program equivalences.

1. *IF And b1 b2 THEN c1 ELSE c2* $\sim$ *IF b1 THEN IF b2 THEN c1 ELSE c2 ELSE c2*
2. *WHILE And b1 b2 DO c* $\sim$ *WHILE b1 DO WHILE b2 DO c*
3. *WHILE And b1 b2 DO c* $\sim$ *WHILE b1 DO c*;; *WHILE And b1 b2 DO c*
4. *WHILE Or b1 b2 DO c* $\sim$ *WHILE Or b1 b2 DO c*;; *WHILE b1 DO c*

Exercise 5.2 Nondeterminism

In this exercise we extend our language with nondeterminism. We will define *nondeterministic choice* (c_1 *OR* c_2), that decides nondeterministically to execute c_1 or c_2; and *assumption* (*ASSUME b*), that behaves like *SKIP* if *b* evaluates to true, and returns no result otherwise.

1. Modify the datatype *com* to include the new commands *OR* and *ASSUME*.
2. Adapt the big step semantics to include rules for the new commands.
3. Prove that c_1 *OR* $c_2 \sim c_2$ *OR* c_1.
4. Prove: (*IF b THEN c1 ELSE c2*) $\sim$ ((*ASSUME b*; *c1*) *OR* (*ASSUME* (*Not b*); *c2*))

Note: It is easiest if you take the existing theories and modify them.

Exercise 5.3 Deskip

Define a recursive function

fun *deskip* :: *“com ⇒ com”*

that eliminates as many *SKIP*s as possible from a command. For example:

deskip (*SKIP*;; *WHILE b DO* (*x* ::= *a*;; *SKIP*)) = *WHILE b DO x* ::= *a*

Prove its correctness by induction on *c*:
Hint: Take a look at *SkipE* and *sim_while_cong*.

lemma *“deskip c ∼ c”*

Homework 5.1 Listing intermediate states

Submission until Monday, November 28, 2022, 23:59pm.

For program analysis tools such as debugger, it is often helpful to list all intermediate states in the execution of a program. Define an inductive predicate *ls*, such that *ls c s ss t* holds iff (*c*, *s*) ⇒ *t* and *ss* are all the intermediate states (we will refine that predicate later):

inductive *ls* :: *“com ⇒ state ⇒ state list ⇒ state ⇒ bool”*

For a simple setup, we declare introduction and elimination rules (feel free to tune this):

declare *ls.intros*[*intro*]
declare *ls.cases*[*elim*]
code_pred *ls* **.**

Show that your predicate is correct w.r.t to big-step semantics. With the right predicate and induction setup, both proofs should be nearly automatic (if they are not, don't spend your time here - we will refine the predicate).

theorem *big_ls*: *“(c,s) ⇒ t ⟹ ∃ sts. ls c s sts t”*
theorem *ls_big*: *“ls c s ss t ⟹ (c,s) ⇒ t”*

You might have wondered which intermediate states to record if the state did *not* change. Use the existing semantics for single steps (→) to infer which states you need to record additionally, and add them to your predicate - this should not break your (⇒) proofs.

A few test cases for the intermediate lists:

abbreviation *“ss_x c s ≡ {map (λs. s ''x'') ss |ss t . ls c s ss t}”*
values *“ss_x (IF Bc True THEN ''x'' ::= N 3 ELSE ''x'' ::= N 1) <>”* — [0, 3]
values *“ss_x (WHILE Less (V ''x'') (N 1) DO ''x'' ::= Plus (V ''x'') (N 1)) <>”* — [0, 0, 1, 1, 1, 1]

Now, we want to prove that the returned list corresponds exactly to the small steps. Start by showing that there is a step from *s* to the head of the list (unless the command is *SKIP*):

lemma *ls_step*: *“⟦ls c s ss t; c ≠ SKIP⟧ ⟹ (case ss of*

$[] \Rightarrow (c,s) \rightarrow (SKIP,t)$
$\mid (x\#\_) \Rightarrow \exists\, c'.\ (c,s) \rightarrow (c',x))$”

That alone is not enough to show our construction correct, we also need a lemma to show that this small step preserves the list semantics:

lemma *ls_ls*: “$[\![ls\ c\ s_1\ (s_2\#ss)\ s_3;\ (c,s_1) \rightarrow (c',s_2)]\!] \Longrightarrow ls\ c'\ s_2\ ss\ s_3$”

Finally, put it all together:

theorem *ls_steps*: “$ls\ c\ s_1\ (ss_1@[s_2]@ss_2)\ t \Longrightarrow \exists\, c'.\ (c,s_1) \rightarrow\!*\ (c',s_2)$”