Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Kevin Kappelmann, Lukas Stevens

WS 24/25
14.11.2024

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 Control Flow Graphs

Submission until Wednesday, November 20, 23:59pm.

In this homework, we want to study the concept of *control flow graphs* for IMP and connect it to the small-step semantics.

A control flow graph is a labeled transition system, where the edges are labeled with *effects*. An effect is a partial function on states, returning *None* when the test for a Boolean condition fails:

type_synonym *effect* = *"state ⇒ state option"*
type_synonym *'q cfg* = *"('q,effect) lts"*

Intuitively, the control flow graph is executed by following a path and applying the effects of the actions to the state. Lift effects to paths. Only paths where all tests succeed shall yield a result ≠ *None*.

fun *eff_list* :: *"effect list ⇒ state ⇒ state option"*

The control flow graph of a WHILE-Program can be defined over nodes that are commands. Complete the following definition. (*Hint:* Have a look at the small-step semantics first)

inductive *cfg* :: *"com cfg"* **where**
  cfg_assign: *"cfg (x ::= a) (λs. Some (s(x:=aval a s))) (SKIP)"*
| *cfg_Seq2*: *"cfg c1 e c1' ⟹ cfg (c1;;c2) e (c1';;c2)"*

We want to show that the effects of paths in the CFG match the small-step semantics. Prove the theorem for a single step first:

theorem *eq_step*: *"(c,s) → (c',s') ⟷ (∃e. cfg c e c' ∧ e s = Some s')"*

Now prove the main theorem:

theorem *eq_path*: *"(c,s) →∗ (c',s') ⟷ (∃π. word cfg c π c' ∧ eff_list π s = Some s')"*

Homework 5.2 Resource management

Submission until Wednesday, November 20, 23:59pm.

Frequently, programs need to allocate resources and clean them up afterwards, even in case of exceptions. Extend IMP with such constructs:

- *THROW* indicates that there is an error
- *ATTEMPT* c_1 *CLEANUP* c_2 executes c_1 until and exception is thrown and always executes c_2.

  The detailed semantics of these constructs are as follows.

  Command *THROW* throws an exception. The only command that can catch an exception is *ATTEMPT* c_1 *CLEANUP* c_2: if an exception is thrown by c_1, execution stops there and continues with c_2. If no exception is thrown, c_2 is also executed. An exception being thrown during c_2 aborts execution of c_2 and propagates "upwards" to the next *ATTEMPT* block.

  Similarly to the small-step semantics, the big-step semantics is now of type *com* × *state* ⇒ *com* × *state*. In a big step $(c,s) \Rightarrow (x,t)$, x is *THROW* if an exception has been thrown, otherwise it is *SKIP*.

  Copy existing types and definitions from *Big_Step* and adapt them.

Step 1 Define the modified big-step semantics.

inductive *big_step* :: *"com × state ⇒ com × state ⇒ bool"* (**infix** *"⇒" 55*)

Step 2 Adapt the previous auxiliary setup from the *BigStep* theory, including rule inversion.

We will also need the introduction & induction rules:

lemmas *big_step_induct* = *big_step.induct*[*split_format*(*complete*)]
declare *big_step.intros*[*intro*]

Step 3 Prove that (⇒) always produces *SKIP* or *THROW*.

lemma *big_step_result*: "$(c,s) \Rightarrow (c',s') \Longrightarrow (c' = SKIP \vee c' = THROW)$"

Step 4 The small-step semantics can also be adjusted. It has the same type as before, but instead of having only *SKIP* as the final command, we can also have *THROW*. Exceptions propagate upwards until an enclosing *ATTEMPT* is found, that is, until a configuration (*ATTEMPT THROW CLEANUP c*, *s*) is reached.

Define the modified small-step semantics and prove that it is complete wrt to the big-step semantics.

inductive *small_step* :: *"com ∗ state ⇒ com ∗ state ⇒ bool"* (**infix** *"→" 55*)

abbreviation *small_steps* :: *"com ∗ state ⇒ com ∗ state ⇒ bool"* (**infix** *"→∗" 55*)
 where *"x →∗ y == star small_step x y"*

declare *small_step.intros*[*simp*,*intro*]

You may need some lemmas from the existing theories. In addition, you might need a new lemma about $x \rightarrow* y$ and $ATTEMPT$.

lemma *big_to_small*: *"cs ⇒ xt ⟹ cs →∗ xt"*