Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Lars Hupel

WS 2017/18
21.11.2017

Semantics of Programming Languages

Exercise Sheet 6

Exercise 6.1 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* ∼ *IF b1 THEN IF b2 THEN c1 ELSE c2 ELSE c2*
2. *WHILE And b1 b2 DO c* ∼ *WHILE b1 DO WHILE b2 DO c*
3. *WHILE And b1 b2 DO c* ∼ *WHILE b1 DO c*;; *WHILE And b1 b2 DO c*
4. *WHILE Or b1 b2 DO c* ∼ *WHILE Or b1 b2 DO c*;; *WHILE b1 DO c*

Exercise 6.2 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*) ∼ ((*ASSUME b*; *c1*) *OR* (*ASSUME* (*Not b*); *c2*))

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

General homework instructions

- All proofs in the homework must be carried out in Isar style.
- You can upload multiple files in the submission interface.

Homework 6.1 Index

Submission until Tuesday, November 28, 10:00am.

Define a function *index_of* that finds the index of an element in a list:

fun *index_of* :: *"'a ⇒ 'a list ⇒ nat option"*

index_of should return *Some i* if the element occurs at the *i*-th position in the list and *None* otherwise.

Prove the following property:

lemma *index_of_prefix*:
  "index_of x xs = Some i ⟹ ∃ ys zs. xs = ys @ x # zs ∧ length ys = i"

Homework 6.2 Fuel your executions

Submission until Tuesday, November 28, 10:00am.

If you try to define a function to execute a program, you will run into trouble with the termination proof (the program might not terminate).

In this exercise, you will define an execution function that tries to execute the program for a bounded number of loop iterations. It gets an additional *nat* argument, called fuel, which decreases for every loop iteration. If the execution runs out of fuel, it stops and returns *None*.

Ultimately, we want to show that some classes of programs (that do not contain *WHILE*) can always be executed to *Some*.

Before working on this exercise, read the entire text carefully. Use the template that is provided on the webpage, so you don't have to copy definitions from the sheet.

Step 1 Define the remaining clauses of the *exec* function:

fun *exec* :: *"com ⇒ state ⇒ nat ⇒ state option"* **where**
  "exec _ s 0 = None"
| *"exec SKIP s f = Some s"*
| *"exec (x::=v) s f = Some (s(x:=aval v s))"*
| *"exec (c1;;c2) s f = (*

case exec c1 s f of
None $\Rightarrow$ *None*
| *Some* s' $\Rightarrow$ *exec c2* s' *f*)"

Your definition should be equivalent to the big-step semantics, i.e.:

theorem *exec_equiv_bigstep*: "($\exists f.$ *exec c s f* = *Some* s') $\longleftrightarrow$ $(c,s) \Rightarrow s'$"

Step 2 (optional, 5 bonus points) Prove the equivalence property. You can find hints in the template.

lemma *exec_imp_bigstep*: "*exec c s f* = *Some* $s' \Longrightarrow (c,s) \Rightarrow s'$"
lemma *bigstep_imp_exec*: "$(c,s) \Rightarrow s' \Longrightarrow \exists k.$ *exec c s k* = *Some* s'"

Step 3 Define a function that returns *True* if a *com* is *While*-free, i.e. contains no *WHILE*:

fun *while_free* :: "*com* $\Rightarrow$ *bool*"

Step 4 Prove that for any while-free program c, there is always a fuel f such that *exec c s f* $\neq$ *None*.
Hint: Construct a small while-free program and try to execute it with *exec*, using various values for fuel.

lemma *while_free_fuel*: "*while_free c* $\Longrightarrow \exists f.$ *exec c s f* $\neq$ *None*"

Homework 6.3 Resource management

Submission until Tuesday, November 28, 10:00am.

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* $\times$ *state* $\Rightarrow$ *com* $\times$ *state*. In a big step $(c,s) \Rightarrow (x,t)$, x is *THROW* if an exception has been thrown, otherwise it is *SKIP*.

Solve this exercise in a separate file that does not import *Big_Step*. Otherwise you will get plenty of ambiguity errors. If necessary, copy existing types and definitions and adapt them in that file.

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.

Hint: Don't forget to declare the introduction & induction rules:

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

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

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

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 $x \to\ast y$ and *ATTEMPT*.

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