Semantics of Programming Languages
Exercise Sheet 6

Exercise 6.1  Program Equivalence

Prove or disprove (by giving counterexamples) the following program equivalences.
1. \( \text{IF And } b_1 \ b_2 \ \text{THEN } c_1 \ \text{ELSE } c_2 \sim \text{IF } b_1 \ \text{THEN } \text{IF } b_2 \ \text{THEN } c_1 \ \text{ELSE } c_2 \ \text{ELSE } c_2 \)
2. \( \text{WHILE And } b_1 \ b_2 \ \text{DO } c \sim \text{WHILE } b_1 \ \text{DO WHILE } b_2 \ \text{DO } c \)
3. \( \text{WHILE And } b_1 \ b_2 \ \text{DO } c \sim \text{WHILE } b_1 \ \text{DO } c;; \text{WHILE And } b_1 \ b_2 \ \text{DO } c \)
4. \( \text{WHILE Or } b_1 \ b_2 \ \text{DO } c \sim \text{WHILE Or } b_1 \ b_2 \ \text{DO } c;; \text{WHILE } b_1 \ \text{DO } c \)

Hint: Use the following definition for \( \text{Or} \):

definition Or :: “bexp ⇒ bexp ⇒ bexp” where
“Or b1 b2 = Not (And (Not b1) (Not b2))”

Exercise 6.2  Nondeterminism

In this exercise we extend our language with nondeterminism. We will define \( \text{nondeterministic choice } (c_1 \ \text{OR } c_2) \), that decides nondeterministically to execute \( c_1 \) or \( c_2 \); and \( \text{assumption } (\text{ASSUME } b) \), that behaves like \( \text{SKIP} \) if \( b \) evaluates to true, and returns no result otherwise.
1. Modify the datatype \( \text{com} \) to include the new commands \( \text{OR} \) and \( \text{ASSUME} \).
2. Adapt the big step semantics to include rules for the new commands.
3. Prove that \( c_1 \ \text{OR } c_2 \sim c_2 \ \text{OR } c_1 \).
4. Prove: \( (\text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2) \sim ((\text{ASSUME } b; c_1) \ \text{OR } (\text{ASSUME } (\text{Not } b); c_2)) \)

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

Homework 6.1  Continue

Submission until Tuesday, December 6, 10:00am.
Note: This homework comes with a template file. You are strongly encouraged to use it!
Your task is to add a continue command to the IMP language. The continue command should skip all remaining commands in the innermost while loop.

The new command datatype is:

```plaintext
datatype com = SKIP
  | Assign vname aexp ("\,:=\:" [1000, 61] 61)
  | Seq com com ("\,:/\:" [60, 61] 60)
  | If bexp com com ("(IF \,:/\,THEN \,:/\,ELSE \,:/\,)" [0, 0, 61] 61)
  | While bexp com ("(WHILE \,:/\,DO \,:/\,)" [0, 61] 61)
  | CONTINUE
```

The idea of the big-step semantics is to return not only a state, but also a continue flag, which indicates that a continue has been triggered. Modify/augment the big-step rules accordingly:

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

Now, write a function that checks that continues only occur in while-loops

```plaintext
fun continue_ok :: "com ⇒ bool"
```

Show that the continue triggered-flag is not set after executing a well-formed command

```plaintext
lemma "[\[(c,s) ⇒ (continue,t); continue_ok c\]] ⇒ ¬continue"
```

In the presence of CONTINUE, some additional sources of dead code arise. We want to eliminate those which can be identified syntactically (that is we do not want to analyze boolean expressions). For instance, the following holds:

```plaintext
lemma "(IF b THEN CONTINUE ELSE CONTINUE;; c) ∼ (CONTINUE)"
```

Write a function elim that eliminates dead code caused by use of CONTINUE. You only need to contract commands because of CONTINUE, you do not need to eliminate SKIPs.

The following should hold for elim:

```plaintext
lemma "elim c ∼ c"
```

Prove this direction:

```plaintext
lemma elim_complete:
  "(c, s) ⇒ (b, s') ⇒ (elim c, s) ⇒ (b, s')"
```

*BONUS:* Also prove the converse direction:
lemma elim_sound:
\[
(\text{elim } c, s) \Rightarrow (b, s') \Rightarrow (c, s) \Rightarrow (b, s')
\]

lemma
\[
\text{"elim } c \sim c"
\]
using elim_sound elim_complete by fast

Homework 6.2 Fuel your executions

Submission until Tuesday, December 6, 10:00am. Note: We provide a template for this homework on the lecture’s homepage.

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 returning None. We will work on the variant of IMP from the first exercise.

fun exec :: “com ⇒ state ⇒ nat ⇒ (bool × state) option” where
\[
\text{exec } s 0 = \text{None}
\]
\[
\text{exec } \text{SKIP } s f = \text{Some } (\text{False}, s)
\]
\[
\text{exec } (x::=v) s f = \text{Some } (\text{False}, s(x:=aval v s))
\]
\[
\text{exec } (c1;;c2) s f = (\text{case exec } c1 s f of}
\]
\[
\text{None ⇒ None}
\]
\[
\text{Some } (\text{True}, s') ⇒ \text{Some } (\text{True}, s')
\]
\[
\text{Some } (\text{False}, s') ⇒ \text{exec } c2 s' f)
\]
\[
\text{exec } (\text{IF } b \text{ THEN } c1 \text{ ELSE } c2) s f =
\]
\[
(\text{if bval } b s \text{ then exec } c1 s f \text{ else exec } c2 s f)
\]
\[
\text{exec } (\text{WHILE } b \text{ DO } c) s (\text{Suc } f) = (\text{if bval } b s \text{ then}
\]
\[
(\text{case } (\text{exec } c s f) \text{ of}
\]
\[
\text{None ⇒ None} | \text{Some } (\text{cont}, s') ⇒ \text{exec } (\text{WHILE } b \text{ DO } c) s' f)\]
\[
\text{else } \text{Some } (\text{False}, s))”
\]
\[
\text{exec } \text{CONTINUE } s f = \text{Some } (\text{True}, s)
\]

Prove that the execution function is correct wrt. the big-step semantics:

theorem exec_equiv_bigstep: “(∃ i. exec c s f = Some s’) ⇔ (c, s) ⇒ s’”

In the following, we give you some guidance for this proof. The two directions are proved separately. The proof of the first direction should be rather straightforward, and is left to you. Recall that is usually best to prove a statement for a (complex) recursive function using its specific induction rule (c.f. sect. 2.3.4 in the book), and that auto can automatically split “case”-expressions using the split attribute (c.f. sect. 2.5.6).
lemma `exec_imp_bigstep`: \[ \text{exec } c \ s \ f = \text{Some } s' \implies (c, s) \Rightarrow s' \]

For the other direction, prove a monotonicity lemma first: If the execution terminates with fuel \( f \), it terminates with the same result using a larger amount of fuel \( f' \geq f \). For this, first prove the following lemma:

lemma `exec_add`: \[ \text{exec } c \ s \ f = \text{Some } s' \implies \text{exec } c \ s \ (f + k) = \text{Some } s' \]

Only the WHILE-case requires some effort. Hint: Make a case distinction on the value of the condition \( b \). You can find the proof for the easy cases in the template.

Now the monotonicity lemma that we want follows easily:

lemma `exec_mono`: \[ \text{exec } c \ s \ f = \text{Some } (\text{brk}, s') \implies f' \geq f \implies \text{exec } c \ s \ f' = \text{Some } (\text{brk}, s') \]

by \((\text{auto simp: exec_add dest: le_Suc_ex})\)

The main lemma is proved by induction over the big-step semantics. Recall the adapted induction rule `big_step_induct` that nicely handles the pattern `big_step (c, s) (brk, s')`. You can find the skip, while-true and if-true cases in the template. The other cases are left to you.

lemma `bigstep_imp_si`: \[ (c, s) \Rightarrow (\text{brk}, s') \implies \exists k. \text{exec } c \ s \ k = \text{Some } (\text{brk}, s') \]

Finally, prove the main theorem of the homework:

theorem `exec_equiv_bigstep`: \[ (\exists k. \text{exec } c \ s \ k = \text{Some } (\text{brk}, s')) \leftrightarrow (c, s) \Rightarrow (\text{brk}, s') \]