Semantics of Programming Languages
Exercise Sheet 5

Exercise 5.1  Program Equivalence

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

1. \[ IF \ And \ b_1 \ b_2 \ THEN \ c_1 \ ELSE \ c_2 \ \sim \ IF \ b_1 \ THEN \ IF \ b_2 \ THEN \ c_1 \ ELSE \ c_2 \ ELSE \ c_2 \]
2. \[ WHILE \ And \ b_1 \ b_2 \ DO \ c \ \sim \ WHILE \ b_1 \ DO \ WHILE \ b_2 \ DO \ c \]
3. \[ WHILE \ And \ b_1 \ b_2 \ DO \ c \ \sim \ WHILE \ b_1 \ DO \ ;; \ WHILE \ And \ b_1 \ b_2 \ DO \ c \]
4. \[ WHILE \ Or \ b_1 \ b_2 \ DO \ c \ \sim \ WHILE \ Or \ b_1 \ b_2 \ DO \ c \ ;; \ WHILE \ b_1 \ DO \ c \]

Hint: Use the following definition for \textit{Or}:

\begin{equation}
\text{definition} \ Or :: \ \text{"bexp} \Rightarrow \text{bexp} \Rightarrow \text{bexp} \ \text{where}
\begin{align*}
\text{"Or} \ b_1 \ b_2 &= \text{Not} (\text{And} (\text{Not} b_1) (\text{Not} b_2))
\end{align*}
\end{equation}

Exercise 5.2  Nondeterminism

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

1. Modify the datatype \textit{com} to include the new commands \textit{OR} and \textit{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: \((IF \ b \ THEN \ c_1 \ ELSE \ c_2) \ \sim \ ((\text{ASSUME} \ b; \ c_1) \ OR \ (\text{ASSUME} \ (\text{Not} b); \ c_2))\)
5. Adapt the small step semantics, and the equivalence proof of big and small step semantics.

Note: It is easiest if you take the existing theories and modify them.
Homework 5.1  Fuel your executions

Submission until Tuesday, November 18, 2014, 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 steps. It gets an additional \texttt{nat} argument, called fuel, which decreases in every step. If the execution runs out of fuel, it stops returning \texttt{None}.

\begin{verbatim}
fun exec :: “com ⇒ state ⇒ nat ⇒ state option” where
  “exec _ s 0 = None”
| “exec SKIP s (Suc f) = Some s”
| “exec (x:=v) s (Suc f) = Some (s(x:=aval v s))”
| “exec (c1;;c2) s (Suc f) = (case (exec c1 s f) of None ⇒ None | Some s’ ⇒ exec c2 s’ f)”
| “exec (IF b THEN c1 ELSE c2) s (Suc f) = (if bval b s then exec c1 s f else exec c2 s f)”
| “exec (WHILE b DO c) s (Suc f) = (if bval b s then (case (exec c s f) of
  None ⇒ None |
  Some s’ ⇒ exec (WHILE b DO c) s’ f)
else Some s)”

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

\begin{verbatim}
theorem exec_equiv_bigstep: “(∃ i. exec c s f = Some s’) ⇔ (c,s) ⇒ s’”
\end{verbatim}

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 quite straightforward, and is left to you.

\begin{verbatim}
lemma exec_imp_bigstep: “exec c s f = Some s’”
\end{verbatim}

For the other direction, prove a monotonicity lemma first: If the execution terminates with fuel \texttt{f}, it terminates with the same result using a larger amount of fuel \texttt{f+k}.

\begin{verbatim}
lemma exec_mono: “exec c s f = Some s’”
proof (induction c s f arbitrary: s’
  rule: exec.induct[case_names None SKIP ASS SEMI IF WHILE])
  — Note: The case_names attribute assigns (new) names to the cases generated by the induction rule, that can then be used with the case - command, as done below.
  case (WHILE b c s i s’) thus ?case

Only the WHILE-case requires some effort. Hint: Make a case distinction on the value of the condition \texttt{b}.

\begin{verbatim}
qed (auto split: option.split option.split_asm)
\end{verbatim}

2
The main lemma is proved by induction over the big-step semantics. Remember the adapted induction rule `big_step_induct` that nicely handles the pattern `big_step (c,s) s'`.

**Lemma**: `bigstep_imp_si`:

```
(c,s) ⇒ s' →∃ k. exec c s k = Some s''
```

**Proof** (induct rule: `big_step_induct`)

We demonstrate the skip, while-true and sequential composition case here. The other cases are left to you!

```isar
case (Skip s) have "exec SKIP s 1 = Some s" by auto
thus ?case by blast
next
case (WhileTrue b s1 c s2 s3)
then obtain f1 f2 where "exec c s1 f1 = Some s2"
and "exec (WHILE b DO c) s2 f2 = Some s3" by auto
with exec_mono[of c s1 f1 s2 f2]
exec_mono[of "WHILE b DO c" s2 f2 s3 f1] have
"exec c s1 (f1+f2) = Some s2"
and "exec (WHILE b DO c) s2 (f2+f1) = Some s3"
by auto
hence "exec (WHILE b DO c) s1 (Suc (f1+f2)) = Some s3"
using (bval b s1) by (auto simp add: add_ac)
thus ?case by blast
next
case (Seq c1 s1 s2 c2 s3)
then obtain f1 f2 where "exec c1 s1 f1 = Some s2" and "exec c2 s2 f2 = Some s3"
by auto
with exec_mono[of c1 s1 f1 s2 f2]
exec_mono[of c2 s2 f2 s3 f1]
have
"exec c1 s1 (f1+f2) = Some s2" and "exec c2 s2 (f2+f1) = Some s3"
by auto
hence "exec (c1;;c2) s1 (Suc (f1+f2)) = Some s3" by (auto simp add: add_ac)
thus ?case by blast
```

Finally, prove the main theorem of the homework:

**Theorem** `exec_equiv_bigstep`:

```
(∃ k. exec c s k = Some s') ↔ (c,s) ⇒ s'
```

---

**Homework 5.2** Skipping over Invisible States

*Submission until Tuesday, November 18, 2014, 10:00am.* Bonus homework, 5 bonus points. Note: This is quite hard, do not waste too much time on it. Partial solutions will be graded.

First, we avoid name clashes with the imp semantics:

hide_const `AExp.V` — Hides the constant, so we can use V as parameter name again

We describe a transition system by a relation over states `step :: 's rel`. Note that `'s rel` is short for `('s×'s)` set.
Intuitively, \((s, s') \in \text{step}\) means that the system can go from state \(s\) to state \(s'\) in one step.

Next, let \(V : \text{\textquotesingle}s\ text{\textquotesingle} set\) be a set of visible states. Define a constant \(\text{skip}\) that performs at least one step, and continues performing steps until a visible state is reached:

\[
\text{inductive_set } \text{skip} :: \quad \text{\textquotesingle}s\ text{\textquotesingle} set \Rightarrow \text{\textquotesingle}s\ rel \Rightarrow \text{\textquotesingle}s\ rel \text{ for } V \text{ step}
\]

For example, let \(v_1, v_2, \ldots\) be visible, and \(i_1, i_2, \ldots\) be invisible states. If \(\text{step}\) admits the steps \(v_1 \rightarrow i_2 \rightarrow i_3 \rightarrow v_4\), then we should have \((v_1, v_4) \in \text{skip} V \text{ step}\) and also \((i_2, v_4) \in \text{skip} V \text{ step}\).

The theories \(\text{Transitive\_Closure}\) and \(\text{Relation}\) provide useful functions to compose step relations. For example, \(\text{step1} \circ \text{step2}\) is the relation that you obtain by first executing a \(\text{step1}\) and then a \(\text{step2}\). Moreover, \(\text{step}^*\) is the reflexive transitive closure. Use \text{find\_theorems} to find some useful lemmas, for example

\[
\text{find\_theorems \textquotesingle op O\textquoteright \ name: \textquotesingle Relation\textquoteright }.
\]

\[
\text{find\_theorems \textquotesingle \_\_\textquoteright \ name: \textquotesingle induct\textquoteright}.
\]

Note: You can also write \(\text{step}\^*\) instead of \(\text{step}^*\), it’s different syntax for the same thing, namely \(\text{rtrancl step}\). In order to get the second form in \text{jEdit}, type \textbackslash<\textsup>*, or use the shortcut \text{CTRL+e UP}.

Your next task is to prove an alternative characterization of \(\text{skip}\) in terms of reflexive transitive closure and function composition. Intuitively, the lemma below states, that \(\text{skip}\) goes one step, then arbitrarily many steps from invisible states (\(-V\) is set complement), and finally ends up in a visible state.

\[
\text{lemma \textquotesingle \text{skip} V \text{ step} = (step O (step} \cap (\{-V\} \times \text{UNIV})^*) \cap (\text{UNIV} \times V)\text{\textquoteright}.
\]

Hints:

- Use Isar where it makes sense.
- Prove the two directions of this lemma separately, bring them into the form \((s, s') \in \ldots \Rightarrow (s, s') \in \ldots\)
- In the \(\Leftarrow\)-direction, the induction on reflexive transitive closure cannot be applied immediately. Bring the statement into a form with an assumption \((a, b) \in (\ldots)^*\) first.
- reflexive transitive closure comes with induction rules for both directions (prepending, appending). Figure out which one you need!
- Hammering on hard nuts sometimes helps to crack them!