Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D., Dr. Peter Lammich
Johannes Hölzl

WS 2014/15
11. 11. 2014, 11:11am

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 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*

Hint: Use the following definition for *Or*:

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

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*))
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 *nat* argument, called fuel, which decreases in every step. If the execution runs out of fuel, it stops returning *None*.

```
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:

```
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 quite straightforward, and is left to you.

```
lemma exec_imp_bigstep: "exec c s f = Some s' ⟹ (c,s) ⇒ 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 $f+k$.

```
lemma exec_mono: "exec c s f = Some s' ⟹ exec c s (f+k) = 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 *b*.

```
qed (auto split: option.split option.split_asm)
```

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!

```
  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, $(s,\ s')$ $\in$ *step* means, that the system can go from state s to state s' in one step.

Next, let V : *'s set* be a set of visible states. Define a constant *skip* that performs at least one step, and continues performing steps until a visible state is reached:

inductive_set *skip* :: *"'s set ⇒ 's rel ⇒ 's rel"* **for** *V step*

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

The theories *Transitive_Closure* and *Relation* provide useful functions to compose step relations. For example, *step1 O step2* is the relation that you obtain by first executing a *step1* and then a *step2*. Moreover, $step^*$ is the reflexive transitive closure. Use *find_theorems* to find some useful lemmas, for example

find_theorems *"op O" name: "Relation."*
find_theorems *"_*" name: "induct"*

Note: You can also write *step*ˆ∗ instead of $step^*$, it's different syntax for the same thing, namely *rtrancl step*. In order to get the second form in jEdit, type `\<^sup>*` , or use the shortcut `CTRL+e UP`.

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

lemma *"skip V step = (step O (step ∩ ((− V) × UNIV))*) ∩ (UNIV × V)"*

Hints:

- Use Isar where it makes sense.
- Prove the two directions of this lemma separately, bring them into the form $(s,s')\in\ldots \implies (s,s')\in\ldots$.
- In the $\Longleftarrow$-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!