Technische Universität München
Fakultät für Informatik
Dr. Peter Lammich
Simon Wimmer

WS 2018/19
06.11.2018

Semantics of Programming Languages

Exercise Sheet 4

Exercise 4.1 Stars and Paths

We want to write down an inductive predicate characterizing paths in directed graphs. We will model directed graphs simply as a relation specifying the edges of type $'v \Rightarrow 'v \Rightarrow bool$. The statement *is_path E u p v* should denote that there is path from u to v in the graph specified by E using the vertices in p. The list of vertices p should contain u as its first element but not v as it last element. Try to write down this predicate:

inductive *is_path* :: *"('v ⇒ 'v ⇒ bool) ⇒ 'v ⇒ 'v list ⇒ 'v ⇒ bool"*

Now also write down a recursive function *path* such that *path E u p v* checks whether p is a valid path from u to v in E:

fun *path* :: *"('v ⇒ 'v ⇒ bool) ⇒ 'v ⇒ 'v list ⇒ 'v ⇒ bool"* **where**

Prove that *path* and *is_path* really represent the same specification of a path:

theorem *path_is_path*:
"path E u xs v ⟷ is_path E u xs v"

It is not possible to write down a function similar to *path* to define the Kleene star. Why? However, we can use a function to define E^n for any natural number n and binary relation E, such that $(u,v) \in E^n$ iff v can be reached from u by exactly n steps in E. Complete its definition:

fun *star_n* :: *"('v ⇒ 'v ⇒ bool) ⇒ nat ⇒ 'v ⇒ 'v ⇒ bool"* **where**

Show the following correspondence between *star_n* and *star*:

theorem *star_n_star*:
"star E u v ⟷ (∃ k. star_n E k u v)"

Try to find a similar correspondence between *star_n* and *is_path* and prove it!

Exercise 4.2 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.

Homework instructions

- All proofs in the homework must be carried out in Isar style. Of course, simple lemmas may be proved in a single line, e.g. *by* (*induction* ...) (*auto simp*: ... *intro*: ... *elim*: ...)

Homework 4.1 Fuel your executions

Submission until Tuesday, November 13, 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 the current state.

Ultimately, we want to show that some classes of programs (that do not contain *WHILE*) can always be executed such that a positive amount of fuel remains at the end.

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.

Consider the following definition of an *exec* function that executes a program with a finite fuel as described above:

```
fun exec :: "com ⇒ state ⇒ nat ⇒ state × nat" where
"exec SKIP s f = (s, f)"
| "exec (x::=v) s f = (s(x:=aval v s), f)"
| "exec (c1;;c2) s f = (
    let (s, f') = exec c1 s f in exec c2 s f')"
| "exec (IF b THEN c1 ELSE c2) s 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
      (let (s, f') = exec c s f in
        if f > 0 then exec (WHILE b DO c) s (min f f') else (s, 0)
      ) else (s, Suc f))"
| "exec _ s 0 = (s, 0)"
```

This definition is equivalent to the big-step semantics, i.e.:

lemma *exec_equiv_bigstep*: "$(\exists f\, f'.\ exec\ c\ s\ f = (s', f') \wedge f' > 0) \longleftrightarrow (c,s) \Rightarrow s'$"

Step 1 Define a function that returns *True* if a *com* is *While*-free, i.e. contains no *WHILE*:

fun *while_free* :: "*com* ⇒ *bool*"

Step 2 Prove that for any while-free program c, there is always a fuel f such that when we start we with f fuel initially, the execution of c will not run out of fuel:

theorem *while_free_fuel*: "*while_free c* $\Longrightarrow \exists f\, s'\, f'.$ *exec c s f* $= (s', f') \wedge f' > 0$"

Hint: You will need an auxiliary lemma stating that fuel does never decrease during an execution. Remember that the modifier *split*: *prod.splits if_splits* for *auto* and friends can come in handy.

Step 3 Prove the equivalence property. You can find hints in the template.

theorem *exec_imp_bigstep*: "*exec c s f* $= (s', f') \Longrightarrow f' > 0 \Longrightarrow (c,s) \Rightarrow s'$"

theorem *bigstep_imp_exec*: "$(c,s) \Rightarrow s' \Longrightarrow \exists f\, f'.\ f' > 0 \wedge$ *exec c s f* $= (s', f')$"

corollary *exec_equiv_bigstep*: "$(\exists f\, f'.$ *exec c s f* $= (s', f') \wedge f' > 0) \longleftrightarrow (c,s) \Rightarrow s'$"
by (*metis bigstep_imp_exec exec_imp_bigstep*)

Homework 4.2 Simple Paths

Submission until Tuesday, November 13, 10:00am.

Prove, that for any path from u to v, there is also a path from u to v that contains each node at most once.

Hint: Theorems *not_distinct_decomp* and *length_induct* may help.

Start the proof of the main theorem like this:

```
lemma path_distinct:
  assumes "is_path E u p v"
  shows "∃p'. distinct p' ∧ is_path E u p' v"
  using assms
```

proof (*induction p rule*: *length_induct*)
  case *step*: (*1 p*)
  note *IH* = *step.IH*
  note *prems* = *step.prems*
  show *?case* **proof** (*cases "distinct p"*)