Technische Universität München
Fakultät für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

WS 2020/21
30.11.2020

Semantics of Programming Languages

Exercise Sheet 5

Exercise 5.1 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 5.2 Deskip

Define a recursive function

fun *deskip* :: *"com ⇒ com"*

that eliminates as many *SKIP*s as possible from a command. For example:

deskip (SKIP;; WHILE b DO (x ::= a;; SKIP)) = WHILE b DO x ::= a

Prove its correctness by induction on *c*:

lemma
  assumes *"(WHILE b DO c, s) ⇒ t"* **and** *"∀ s t. (c, s) ⇒ t ⟶ (c′, s) ⇒ t"*
    shows *"(WHILE b DO c′, s) ⇒ t"*
lemma *"deskip c ∼ c"*

Exercise 5.3 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. If you work in this template, remove the old *big_step* notations first:

no_notation *Assign* (“_ ::= _” [1000, 61] 61)
no_notation *Seq* (“_;;/ _” [60, 61] 60)
no_notation *If* (“(IF _/ THEN _/ ELSE _)” [0, 0, 61] 61)
no_notation *While* (“(WHILE _/ DO _)” [0, 61] 61)
no_notation *big_step* (**infix** “⇒” 55)
no_notation *equiv_c* (**infix** “∼” 50)

Homework 5.1 Break

Submission until Sunday, Dec 6, 23:59.

Your task is to add a break command to the IMP language. The break may be used in a while loop, and it should immediately exit the loop.

The new command datatype is:

```
datatype
com = Skip                  ("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)
  | Break                   ("BREAK")
```

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

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

Add proof automation as in *HOL−IMP.Big_Step*:

```
declare big_step.intros [intro]

lemmas big_step_induct = big_step.induct[split_format(complete)]

inductive_cases SkipE[elim!]: "(SKIP,s) ⇒ t"
inductive_cases BreakE[elim!]: "(BREAK,s) ⇒ t"
inductive_cases AssignE[elim!]: "(x ::= a,s) ⇒ t"
inductive_cases SeqE[elim!]: "(c1;;c2,s1) ⇒ s3"
inductive_cases IfE[elim!]: "(IF b THEN c1 ELSE c2,s) ⇒ t"
inductive_cases WhileE[elim]: "(WHILE b DO c,s) ⇒ t"

lemma assign_simp:
  "(x ::= a,s) ⇒ (brk,s′) ⟷ (s′ = s(x := aval a s) ∧ ¬brk)"
  by auto
```

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

```
fun break_ok :: "com ⇒ bool"
```

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

```
theorem ok_brk: "⟦(c, s) ⇒ (brk, t); break_ok c⟧ ⟹ ¬brk"
```

In the presence of *BREAK*, some additional sources of dead code arise. We want to eliminate those which can be identified syntactically (that is, without analyzing boolean expressions).

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

```
fun elim :: "com ⇒ com"
```

Now prove correctness for *elim*:

```
abbreviation equiv_c :: "com ⇒ com ⇒ bool" (infix "∼" 50) where
  "c ∼ c′ ≡ (∀ s t. (c, s) ⇒ t  =  (c′, s) ⇒ t)"

theorem elim_complete: "(c, s) ⇒ (b, s′) ⟹ (elim c, s) ⇒ (b, s′)"
theorem elim_sound: "(elim c, s) ⇒ (b, s′) ⟹ (c, s) ⇒ (b, s′)"
lemma "elim c ∼ c"
  using elim_sound elim_complete by fast
```

Homework 5.2 Fuel your executions

Submission until Sunday, Dec 6, 23:59.

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).

To overcome this, 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 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. Make sure that the *big_step_test* on the submission system works before starting this exercise!

```
fun exec :: "com ⇒ state ⇒ nat ⇒ (bool × state) option" where
value "(case (
    exec (
      WHILE (Bc True) DO
      IF (Less (V ''x'') (N 4))
        THEN ''x''::= (Plus (V ''x'') (N 1))
         ELSE BREAK
    ) <> 10
  ) of (Some (False, s)) ⇒
    s ''x''
  ) = 4"
```

We want to prove that the execution function is correct wrt. the big-step semantics.

In the following, we give you some guidance for this proof. The two directions are proved separately. The proof of the first direction is left to you. Recall that is usually best to prove a statement for a (complex) recursive function using its specific induction rule, and that auto can automatically split "case"-expressions using the *split* attribute.

```
theorem 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 $f' \geq f$. For this, first prove the following lemma:

```
theorem exec_add: "exec c s f = Some s' ⟹ exec c s (f + k) = Some s'"
```

Now the monotonicity lemma that we want follows easily:

```
lemma exec_mono: "exec c s f = Some (brk, s') ⟹ f' ≥ f ⟹ exec c s f' = Some (brk, s')"
  by (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'*).

```
theorem bigstep_imp_si:
  "(c,s) ⇒ (brk, s') ⟹ ∃k. exec c s k = Some (brk, s')"
proof (induct rule: big_step_induct)
```

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

```
  case (Skip s) have "exec SKIP s 1 = Some (False, s)" by auto
  thus ?case by blast
next
```

```
  case (WhileTrue b s1 c s2 brk3 s3)
  then obtain f1 f2 where "exec c s1 f1 = Some (False, s2)"
    and "exec (WHILE b DO c) s2 f2 = Some (brk3, s3)" by auto
  with exec_mono[of c s1 f1 False s2 "max f1 f2"]
    exec_mono[of "WHILE b DO c" s2 f2 brk3 s3 "max f1 f2"] have
    "exec c s1 (max f1 f2) = Some (False, s2)"
    and "exec (WHILE b DO c) s2 (max f1 f2) = Some (brk3, s3)"
    by auto
  hence "exec (WHILE b DO c) s1 (Suc (max f1 f2)) = Some (brk3, s3)"
    using ‹bval b s1› by (auto simp add: add_ac)
  thus ?case by blast
next
  case (IfTrue b s c1 brk' t c2)
  then obtain k where "exec c1 s k = Some (brk', t)" by blast
  hence "exec (IF b THEN c1 ELSE c2) s k = Some (brk', t)"
  using ‹bval b s› by (cases k) auto
  thus ?case by blast
next
```

Finally, the main theorem of the homework follows:

```
lemma "(∃k. exec c s k = Some (brk, s')) ⟷ (c,s) ⇒ (brk, s')"
  by (metis exec_imp_bigstep bigstep_imp_si)
```