Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch, Kevin Kappelmann

WS 23/24
21.11.2023

# 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 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*) ∼ ((*ASSUME b*; *c1*) *OR* (*ASSUME* (*Not b*); *c2*))

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

## Exercise 5.3 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*:
Hint: Take a look at *SkipE* and *sim_while_cong*.

**lemma** *"deskip c ∼ c"*

## Homework 5.1 SWITCHeroo

*Submission until Monday, November 28, 23:59pm.*

We introduce the type of SWITCH-IMP programs by modifying the type of *com*-programs as follows:

1. All boolean expressions are replaced by corresponding arithmetic expressions. An an integer should be interpreted as false if it is equal to *0* and true otherwise:
   *is_false x* ≡ *x* = *0*
   *is_true x* ≡ ¬ *is_false x*
2. We remove the *IF b THEN c1 ELSE c2* constructor.
3. We add a switch constructor *SWITCH a CASES acs*. The constructor takes an arithmetic expression *a* and a list of pairs *acs* of arithmetic expressions and commands. The semantics of *SWITCH a CASES acs* is as follows:
   - Switching on an empty list does not change the state.
   - Switching on a non-empty list (*a′*,*c*) # *acs* executes *c* and then exits the switch if *a* and *a′* are semantically equal; otherwise, it continues switching on *a* and *acs*.

**datatype** *scom* = *sSKIP* | *sAssign* (*char list*) *aexp* | *sSeq scom scom* | *sWhile aexp scom* | *sSwitch aexp* ((*aexp* × *scom*) *list*)

Define the big-step semantics as an inductive predicate. Hint: Do not use list functions!

**inductive** *sbig_step* :: *"scom × state ⇒ state ⇒ bool"* (**infix** *"⇒s" 55*)

Define a function that compiles SWITCH-IMP programs to semantically equivalent IMP programs.

**fun** *scom_to_com* :: *"scom ⇒ com"*

Prove that your compiler is correct:
Hint: if one of the parameters of your induction is a complex term *t* (i.e. not just a variable), you can pass the term as an argument to the induction method as you would

do for a variable, but you need to enclose the term in quotes. The free variables in $t$ must be marked as *arbitrary:*.

**lemma** *big_step_scom_to_com_if_big_step*: "$(c,\ s) \Rightarrow s\ t \Longrightarrow (scom\_to\_com\ c,\ s) \Rightarrow t$"
**lemma** *sbig_step_if_big_step_scom_to_com*: "$(scom\_to\_com\ c,\ s) \Rightarrow t \Longrightarrow (c,\ s) \Rightarrow s\ t$"
**corollary** "$(c,\ s) \Rightarrow s\ t \longleftrightarrow (scom\_to\_com\ c,\ s) \Rightarrow t$"

Now define the single-step small-step semantics as an inductive predicate. Do not use list functions!

**inductive** *ssmall_step* :: "$scom * state \Rightarrow scom * state \Rightarrow bool$" (**infix** "$\rightarrow s$" 55)

As usual, we can take the reflexive, transitive closure of the single-step small-step semantics.

**abbreviation** *ssmall_steps* :: "$scom * state \Rightarrow scom * state \Rightarrow bool$" (**infix** "$\rightarrow s*$" 55)
**where** "$x \rightarrow s*\ y \equiv star\ ssmall\_step\ x\ y$"

Prove the equivalence of the small-step and big-step semantics.

Hint: copy and adapt the lemmas that where proven in the lecture for small-step semantics.

**lemma** *ssmall_steps_if_sbig_step*: "$(c,\ s) \Rightarrow s\ t \Longrightarrow (c,\ s) \rightarrow s*\ (sSKIP,\ t)$"
**lemma** *sbig_step_if_ssmall_steps*: "$cs \rightarrow s*\ (sSKIP,\ t) \Longrightarrow cs \Rightarrow s\ t$"
**corollary** "$(c,\ s) \Rightarrow s\ t \longleftrightarrow (c,\ s) \rightarrow s*\ (sSKIP,\ t)$"

Finally, we consider a non-deterministic, alternative semantics for switches:

- Switching on an empty list does not change the state.
- Switching on a non-empty list $(a',c)\ \#\ acs$ continues switching on $a$ and $acs$ if $a$ and $a'$ are not semantically equal; otherwise, it either first executes $c$ and then exits the switch *or* it continues switching on $a$ and $acs$.

**inductive** *nsbig_step* :: "$scom \times state \Rightarrow state \Rightarrow bool$" (**infix** "$\Rightarrow ns$" 55)

Prove that the non-deterministic semantics can simulate the deterministic semantics.

**lemma** *nsbig_step_if_sbig_step*: "$(c,\ s) \Rightarrow s\ t \Longrightarrow (c,\ s) \Rightarrow ns\ t$"

Prove *or* disprove the converse direction. If you think that one of the following two lemmas is wrong, close the lemma statement with an **oops**.

**lemma** *conjecture_true*: "$(c,\ s) \Rightarrow ns\ t \Longrightarrow \exists\, t.\ (c,\ s) \Rightarrow ns\ t \wedge (c,\ s) \Rightarrow s\ t$"

**lemma** *conjecture_false*: "$\exists\, c\ s\ t.\ (c,\ s) \Rightarrow ns\ t \wedge \neg(\exists\, t.\ (c,\ s) \Rightarrow ns\ t \wedge (c,\ s) \Rightarrow s\ t)$"