Semantics of Programming Languages
Exercise Sheet 7

Exercise 7.1 Deskip

Define a recursive function

\[
\text{fun deskip :: "com ⇒ com"}
\]

that eliminates as many SKIPs as possible from a command. For example:

\[
deskip (\text{SKIP};; \text{WHILE b DO (} x ::= a;; \text{SKIP}) ) = \text{WHILE b DO } x ::= a
\]

Prove its correctness by induction on \( c \):

\[
\text{lemma "deskip } c \sim c"
\]

Exercise 7.2 Small step pre-order

We define a pre-order \( \preceq \) on programs that uses the small-step semantics. The relation \( p \preceq p' \) shall hold if \( p' \) computes for any input the same output as \( p \), and in at most the same number of steps.

The following relation is the \( n \)-steps reduction relation:

\[
\text{inductive } nsteps :: "com * state ⇒ nat ⇒ com * state ⇒ bool"
\]

\[
("\rightarrow\) " [60,1000,60]999)
\]

where

\[
\text{zero_step: "} cs \rightarrow \hat{0} cs"
\]

\[
\text{one_step: "} cs \rightarrow cs' \Rightarrow cs' \rightarrow \hat{n} cs'' \Rightarrow cs \rightarrow \hat{(Suc n)} cs''"
\]

Prove the following lemmas:

\[
\text{lemma small_steps_n: "} cs \rightarrow* cs' \Rightarrow (\exists n. cs \rightarrow \hat{n} cs')"
\]

\[
\text{lemma nsmall_steps: "} cs \rightarrow \hat{n} cs' \Rightarrow cs \rightarrow* cs''"
\]

\[
\text{lemma nsteps_trans: "} cs \rightarrow \hat{n1} cs' \Rightarrow cs' \rightarrow \hat{n2} cs'' \Rightarrow cs \rightarrow \hat{(n1+n2)} cs''"
\]

The pre-order relation is defined as follows:

\[
\text{definition } small_step_pre :: "com ⇒ com ⇒ bool" (\textbf{infix "≤" 50}) \textbf{where}
\]
Prove the following lemma:

**Lemma small_eqv_implies_big_eqv:**

**Assumes** “$c \preceq c'$” “$c' \preceq c$”

**Shows** “$c \sim c'$”

**Exercise 7.3 Compiler optimization**

A common programming idiom is $IF \ b \ THEN \ c$, i.e., the else-branch consists of a single $SKIP$ command.

1. Look at how the program $IF \ Less \ (V \ "x") \ (N \ 5) \ THEN \ "y" ::= N \ 3 \ ELSE \ SKIP$ is compiled by $ccomp$ and identify a possible compiler optimization.
2. Implement an optimized compiler (by modifying $ccomp$) which reduces the number of instructions for programs of the form $IF \ b \ THEN \ c$.
3. Extend the proof of $comp\_bigstep$ to your modified compiler.

**Homework 7.1 Compiling REPEAT**

*Submission until Tuesday, December 2, 10:00am.*

We extend $com$ with a $REPEAT \ c \ UNTIL \ b$ statement, adding the following rules to our big-step semantics:

- **RepeatTrue:** $\llbracket (c, s_1) \Rightarrow s_2; \ bval \ b \ s_2 \rrbracket \Longrightarrow (REPEAT \ c \ UNTIL \ b, s_1) \Rightarrow s_2$
- **RepeatFalse:** $\llbracket (c, s_1) \Rightarrow s_2; \ \neg \ bval \ b \ s_2; \ (REPEAT \ c \ UNTIL \ b, s_2) \Rightarrow s_3 \rrbracket \Longrightarrow (REPEAT \ c \ UNTIL \ b, s_1) \Rightarrow s_3$

Building on this, extend the compiler $ccomp$ and its correctness theorem $ccomp\_bigstep$ to $REPEAT$ loops. **Hint:** the recursion pattern of the big-step semantics and the compiler for $REPEAT$ should match.

Download the files $Repeat\_Big\_Step.thy$ and $Repeat\_Compiler\_Template.thy$. Finish the definition of $ccomp$ and the proof of $ccomp\_bigstep$ in $Repeat\_Compiler\_Template.thy$, and submit this theory using as filename the schema $FirstnameLastname2.thy$.

**Homework 7.2 Commuting sequences of commands**

*Submission until Tuesday, December 13, 10:00am.*

Write a function that collects all variables that occur in a command. (Hint: You need to write such functions also for boolean and arithmetic expressions)

**fun vars ::** “$\text{com} \Rightarrow v\text{name set}$” **where**
Then show the following two lemmas:

**Lemma aval_equiv:**

\[(c, s) \Rightarrow t \Rightarrow \text{vars}_a \cap \text{vars}_c = \{\} \Rightarrow \text{aval}_a t = \text{aval}_a s\]

**Lemma bval_equiv:**

\[(c, s) \Rightarrow t \Rightarrow \text{vars}_b \cap \text{vars}_c = \{\} \Rightarrow \text{bval}_b t = \text{bval}_b s\]

Finally prove that a sequence of commands can be commuted if the commands do not share any common variables:

**Lemma Seq_commute:**

assumes “\(\text{vars}_c1 \cap \text{vars}_c2 = \{\}\)"

shows “\(c1;;c2 \sim c2;;c1\)”

*oops*

One possible way to get there, is to prove the following auxiliary lemma first:

**Lemma Seq_commute’:**

assumes “\((c1, s) \Rightarrow s’\) “\((c2, s’) \Rightarrow t\)” “\(\text{vars}_c1 \cap \text{vars}_c2 = \{\}\)"

shows “\((c2;;c1, s) \Rightarrow t\)”

You only need to do the cases for while-loops and assignment. The latter may necessitate another helper lemma.

**Lemma Seq_commute:**

assumes “\(\text{vars}_c1 \cap \text{vars}_c2 = \{\}\)"

shows “\(c1;;c2 \sim c2;;c1\)”

### Homework 7.3 Algebra of Commands

*Submission until Tuesday, December 13, 10:00am.*

We define an extension of the language with parallel composition (\(\parallel\)) for which we consider the small-step equivalence.

Your task will be to prove various algebraic laws for the small-step equivalence. The most helpful methods will be number induction and/or pair-based rule induction over the \(n\text{steps}\) relation, using \(n\text{steps}_\text{induct}\) (provided below).

**Datatype**

\[\text{com} =\]

| — sequential part as before — |
| \(\text{Par com com}\) (infix “\(\parallel\)” 59) |

**Inductive**

\[\text{small\_step :: “com \ast state \Rightarrow com \ast state \Rightarrow bool” (infix “\(\rightarrow\)” 55)}\]

**Where**

| — sequential part as before — |
| \(\text{ParL: “}(c1, s) \Rightarrow (c1’, s’) \Rightarrow (c1 \parallel c2, s) \Rightarrow (c1’ \parallel c2, s’)”}\ |
| \(\text{ParLSkip: “}(\text{SKIP} \parallel c, s) \Rightarrow (c, s)”}\ |
| \(\text{ParR: “}(c2, s) \Rightarrow (c2’, s’) \Rightarrow (c1 \parallel c2, s) \Rightarrow (c1 \parallel c2’, s’)”}\ |
ParRSkip: “(c || SKIP, s) → (c, s)”

**lemmas** small_step_induct = small_step.induct[split_format(complete)]

**inductive**

nsteps :: “com * state ⇒ nat ⇒ com * state ⇒ bool”

(“_ → _” [60,1000,60,999])

**where**

zero_steps[simp,intro]: “cs → ˆ0 cs” |

one_step[intro]: “cs → cs’ ⇒ cs’ → ˆn cs’’ ⇒ cs → ˆ(Suc n) cs’’”

**lemmas** nsteps_induct = nsteps.induct[split_format(complete)]

**definition**

small_step.pre :: “com ⇒ com ⇒ bool” (infix “≤” 50) where

“c ≤ c’ ≡ (∀ s t n. (c, s) → ˆn (SKIP, t) → (3 n’ ≥ n. (c’, s) → ˆn’ (SKIP, t)))”

Based on the pre-order on programs, define an equivalence relation ≈ on programs.

Now prove commutativity and associativity of ||. You are free to do either automatic or Isar proofs.

**lemma** Par_commute: “c || d ≈ d || c”

**lemma** Par_assoc: “(c || d) || e ≈ c || (d || e)”