Exercise 9.1  Independence analysis

In this exercise you first prove that the execution of a command only depends on its used (i.e., read or assigned) variables. Then you use this to prove commutativity of sequential composition.

Note that this development is entirely done on the small-step semantics.

First show that arithmetic and boolean expressions only depend on the variables occuring in them

\[\text{lemma [simp]}: \text{"}s_1 = s_2 \text{ on } X \Rightarrow \text{vars } a \subseteq X \Rightarrow \text{aval } a s_1 = \text{aval } a s_2\text{"} \]

\[\text{lemma [simp]}: \text{"}s_1 = s_2 \text{ on } X \Rightarrow \text{vars } b \subseteq X \Rightarrow \text{bval } b s_1 = \text{bval } b s_2\text{"} \]

Next, show that executing a command does not invent new variables

\[\text{lemma \text{vars subsetD}[dest]}: \text{"}(c, s) \rightarrow (c', s') \Rightarrow \text{vars } c' \subseteq \text{vars } c\text{"} \]

And that the effect of a command is confined to its variables

\[\text{lemma small_step_confinement}: \text{"}(c, s) \rightarrow (c', s') \Rightarrow s = s' \text{ on } \text{UNIV} - \text{vars } c\text{"} \]

\[\text{lemma small_steps_confinement}: \text{"}(c, s) \rightarrow^* (c', s') \Rightarrow s = s' \text{ on } \text{UNIV} - \text{vars } c\text{"} \]

Hint: These proofs should go through (mostly) automatically by induction.

Now, we are ready to show that commands only depend on the variables they use:

\[\text{lemma small_step_indep}: \text{"}(c, s) \rightarrow (c', s') \Rightarrow s = t \text{ on } X \Rightarrow \text{vars } c \subseteq X \Rightarrow \exists t'. (c, t) \rightarrow (c', t') \land s' = t' \text{ on } X\text{"} \]

\[\text{lemma small_steps_indep}: \text{"}[ (c, s) \rightarrow^* (c', s'); s = t \text{ on } X; \text{vars } c \subseteq X ] \Rightarrow \exists t'. (c, t) \rightarrow^* (c', t') \land s' = t' \text{ on } X\text{"} \]

Two lemmas that may prove useful for the next proof.

\[\text{lemma small_steps(SeqE)}: \text{"}[ (c_1 ; ; c_2, s) \rightarrow^* (\text{SKIP}, s') \Rightarrow \exists t. (c_1, s) \rightarrow^* (\text{SKIP}, t) \land (c_2, t) \rightarrow^* (\text{SKIP}, s') ] \]

by (induction “c1 ; ; c2” s SKIP s’ arbitrary: c1 c2 rule: star_induct)

(blast intro: star_step)

\[\text{lemma small_steps(SeqI)}: \text{"}[ (c_1, s) \rightarrow^* (\text{SKIP}, s'); (c_2, s') \rightarrow^* (\text{SKIP}, t) ] \]
As we operate on the small-step semantics we also need our own version of command equivalence. Two commands are equivalent iff a terminating run of one command implies a terminating run of the other command. And, of course the terminal state needs to be equal when started in the same state.

**definition** equiv_com :: “com ⇒ com ⇒ bool” (infix “∼” 50) where
“c1 ∼ s c2 ⊢⊥ (∀s t. (c1, s) ⇒∗ (SKIP, t) ⊢⊥ (c2, s) ⇒∗ (SKIP, t))”

Show that we defined an equivalence relation

**lemma** ec_refl[simp]: “equiv_com c c”
**lemma** ec_sym: “equiv_com c1 c2 ⊢⊥ equiv_com c2 c1 ”
**lemma** ec_trans[trans]: “equiv_com c1 c2 ⇒ equiv_com c2 c3 ⇒ equiv_com c1 c3”

Note that our small-step equivalence matches the big-step equivalence

**lemma** “c1 ∼ s c2 ⊢⊥ c1 ∼ c2” unfolding equiv_com_def by (metis big_iff_small)

Finally, show that commands that share no common variables can be re-ordered

**theorem** Seq_equiv(Seq_reorder):
  assumes vars: “vars c1 ∩ vars c2 = {}”
  shows “(c1 ;; c2) ∼ s (c2 ;; c1)”
  proof –
  
  As the statement is symmetric, we can take a shortcut by only proving one direction:
  
  fix c1 c2 s t
  assume Seq: “(c1 ;; c2, s) ⇒∗ (SKIP, t)” and vars: “vars c1 ∩ vars c2 = {}”
  have “(c2 ;; c1, s) ⇒∗ (SKIP, t)”
  } with vars show ?thesis unfolding equiv_com_def by (metis Int.commute)
  qed

**Homework 9.1** Available Definitions

Submission until Tuesday, December 16, 10:00am. An available definitions analysis determines which previous assignments x := a are valid equalities x = a at later program points. For example, after x := y+1 the equality x = y+1 is available, but after x := y+1; y := 2 the equality x = y+1 is no longer available. The motivation for the analysis is that if x = a is available before v := a then v := a can be replaced by v := x.

Define an available definitions analysis as a gen/kill analysis, for suitably defined gen and kill (which may need to be mutually recursive):

**fun** gen :: “com ⇒ (vname * aexp) set”
and “kill” :: “com ⇒ (vname * aexp) set” where

definition AD :: “(vname * aexp) set ⇒ com ⇒ (vname * aexp) set” where
“AD A c = gen c ∪ (A − kill c)”

The defining equations for both gen and kill follow the where and are separated by “|” as usual.

A call AD A c should compute the available definitions after the execution of c assuming that the definitions in A are available before the execution of c.

Prove correctness of the analysis:

theorem “[[ (c,s) ⇒ s’; ∀ (x,a) ∈ A. s x = aval a s ]] ⇒ ∀ (x,a) ∈ AD A c. s’ x = aval a s’”

Homework 9.2 Knaster-Tarski Fixed Point Theorem

Submission until Tuesday, December 16, 10:00am.

The Knaster-Tarski theorem tells us that for each set P of fixed points of a monotone function f we have a fixpoint of f which is a greatest lower bound of P. In this exercise, we want to prove the Knaster-Tarski theorem.

First we give a construction of the greatest lower bound of all fixed points P of the function f. This is the union of all sets u smaller than P and f u. Then the task is to show that this is a fixed point, and that it is the greatest lower bound of all sets in P.

Let’s define Inf_fixp:

definition Inf_fixp :: “'a set ⇒ 'a set ⇒ 'a set set ⇒ 'a set” where
“Inf_fixp f P = ⋃ {u. u ⊆ ⋂ P ∩ f u}”

To work directly with this definition is a little cumbersome, we propose to use the following two theorems:

lemma Inf_fixp_upperbound: “X ⊆ ⋂ P ⇒ X ⊆ f X ⇒ X ⊆ Inf_fixp f P”
  by (auto simp: Inf_fixp_def)

lemma Inf_fixp_least: “(⋀u. u ⊆ ⋂ P ⇒ u ⊆ f u ⇒ u ⊆ X) ⇒ Inf_fixp f P ⊆ X”
  by (auto simp: Inf_fixp_def)

Now prove, that Inf_fixp is actually a fixed point of f. IMPORTANT: Only structured Isar proofs are accepted. You are allowed to use auto, etc., but no metis calls (e.g. like the ones generated by sledgehammer).

Hint: First prove Inf_fixp f P ⊆ f (Inf_fixp f P), this will be used for the other direction. It may be helpful to first think about the structure of your proof using pen-and-paper and then translate it into Isar.

lemma Inf_fixp:
  assumes f: “mono f”
assumes $P$: \( \forall p \in P \implies f p = p \)

shows \( \text{Inf}_{\text{fixp}} f P = f (\text{Inf}_{\text{fixp}} f P) \)

Now we prove that it is a lower bound:

lemma \( \text{Inf}_{\text{fixp}} \text{lower} \): \( \text{Inf}_{\text{fixp}} f P \subseteq \bigcap P \)

And that it is the greatest lower bound:

lemma \( \text{Inf}_{\text{fixp}} \text{greatest} \):

assumes \( f q = q \) \( q \subseteq \bigcap P \) shows \( q \subseteq \text{Inf}_{\text{fixp}} f P \)