Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D., Dr. Peter Lammich
Johannes Hölzl

WS 2014/15
9. 12. 2014

Semantics of Programming Languages

Exercise Sheet 9

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

lemma [*simp*]: *"s1 = s2 on X ⟹ vars a ⊆ X ⟹ aval a s1 = aval a s2"*

lemma [*simp*]: *"s1 = s2 on X ⟹ vars b ⊆ X ⟹ bval b s1 = bval b s2"*

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

lemma *vars_subsetD*[*dest*]: *"(c, s) → (c′, s′) ⟹ vars c′ ⊆ vars c"*

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

lemma *small_step_confinement*: *"(c, s) → (c′, s′) ⟹ s = s′ on UNIV − vars c"*
lemma *small_steps_confinement*: *"(c, s) →∗ (c′, s′) ⟹ s = s′ on UNIV − vars c"*

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:

lemma *small_step_indep*:
"(c, s) → (c′, s′) ⟹ s = t on X ⟹ vars c ⊆ X ⟹ ∃ t′. (c, t) → (c′, t′) ∧ s′ = t′ on X"
lemma *small_steps_indep*: *"⟦(c, s) →∗ (c′, s′); s = t on X; vars c ⊆ X⟧*
⟹ ∃ t′. (c, t) →∗ (c′, t′) ∧ s′ = t′ on X"

Two lemmas that may prove useful for the next proof.

lemma *small_steps_SeqE*: *"(c1 ;; c2, s) →∗ (SKIP, s′)*
⟹ ∃ t. (c1, s) →∗ (SKIP, t) ∧ (c2, t) →∗ (SKIP, s′)"
by (*induction "c1 ;; c2" s SKIP s′ arbitrary*: *c1 c2 rule*: *star_induct*)
(*blast intro*: *star.step*)

lemma *small_steps_SeqI*: *"⟦(c1, s) →∗ (SKIP, s′); (c2, s′) →∗ (SKIP, t)⟧*

$\Longrightarrow (c1\ ;;\ c2,\ s) \rightarrow* (SKIP,\ t)$"
by (*induction c1 s SKIP s' rule*: *star_induct*)
(*auto intro*: *star.step*)

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** "$\sim_s$" 50) **where**
"*c1* $\sim_s$ *c2* $\longleftrightarrow$ (∀ *s t*. (*c1*, *s*) →* (*SKIP*, *t*) $\longleftrightarrow$ (*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* $\longleftrightarrow$ *equiv_com c2 c1* "
lemma *ec_trans*[*trans*]: "*equiv_com c1 c2* $\Longrightarrow$ *equiv_com c2 c3* $\Longrightarrow$ *equiv_com c1 c3*"

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

lemma "*c1*$\sim_s$*c2* $\longleftrightarrow$ *c1*$\sim$*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*) $\sim_s$ (*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* $\Rightarrow$ *(vname* $*$ *aexp) set"* **where**

definition *AD* :: *"(vname* $*$ *aexp) set* $\Rightarrow$ *com* $\Rightarrow$ *(vname* $*$ *aexp) set"* **where**
"AD A c = gen c $\cup$ *(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 *"*$[\![(c,s) \Rightarrow s'; \ \forall\ (x,a) \in A.\ s\ x = aval\ a\ s\ ]\!]$
$\Longrightarrow \forall\ (x,a) \in AD\ A\ c.\ s'\ x = aval\ a\ s'$*"*

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.

Lets define *Inf_fixp*:

definition *Inf_fixp* :: *"('a set* $\Rightarrow$ *'a set)* $\Rightarrow$ *'a set set* $\Rightarrow$ *'a set"* **where**
"Inf_fixp f P = $\bigcup \{ u.\ u \subseteq \bigcap P \cap f\ u \}$*"*

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

lemma *Inf_fixp_upperbound*: *"*$X \subseteq \bigcap P \Longrightarrow X \subseteq f\ X \Longrightarrow X \subseteq$ *Inf_fixp f P"*
by (*auto simp*: *Inf_fixp_def*)

lemma *Inf_fixp_least*: *"*$(\bigwedge u.\ u \subseteq \bigcap P \Longrightarrow u \subseteq f\ u \Longrightarrow u \subseteq X) \Longrightarrow$ *Inf_fixp f P* $\subseteq X$*"*
by (*auto simp*: *Inf_fixp_def*)

Now prove, that *Inf_fixp* is acually 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* $\subseteq f$ (*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: "$\bigwedge p.\ p \in P \Longrightarrow f\ p = p$"
shows "*Inf_fixp f P* $= f$ (*Inf_fixp f P*)"

Now we prove that it is a lower bound:

lemma *Inf_fixp_lower*: "*Inf_fixp f P* $\subseteq \bigcap P$"

And that it is the greatest lower bound:

lemma *Inf_fixp_greatest*:
assumes "$f\ q = q$" "$q \subseteq \bigcap P$" **shows** "$q \subseteq$ *Inf_fixp f P*"