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

WS 2013/14
10. 12. 2013

Semantics of Programming Languages

Exercise Sheet 8

Exercise 8.1 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

term "$s = t$ *on* X"

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

lemma [*simp*]: "$s1 = s2$ *on* $X \Longrightarrow$ *vars* $a \subseteq X \Longrightarrow$ *aval* a $s1$ = *aval* a $s2$"

lemma [*simp*]: "$s1 = s2$ *on* $X \Longrightarrow$ *vars* $b \subseteq X \Longrightarrow$ *bval* b $s1$ = *bval* b $s2$"

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

lemma *vars_subsetD*[*dest*]: "$(c, s) \rightarrow (c', s') \Longrightarrow$ *vars* $c' \subseteq$ *vars* c"

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

lemma *small_step_confinement*: "$(c, s) \rightarrow (c', s') \Longrightarrow s = s'$ *on UNIV* $-$ *vars* c"
lemma *small_steps_confinement*: "$(c, s) \rightarrow* (c', s') \Longrightarrow s = s'$ *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*:
"$(c, s) \rightarrow (c', s') \Longrightarrow s = t$ *on* $X \Longrightarrow$ *vars* $c \subseteq X \Longrightarrow \exists t'. (c, t) \rightarrow (c', t') \wedge s' = t'$ *on* X"
lemma *small_steps_indep*: "$[\![(c, s) \rightarrow* (c', s'); s = t$ *on* X; *vars* $c \subseteq X]\!]$
$\Longrightarrow \exists t'. (c, t) \rightarrow* (c', t') \wedge s' = t'$ *on* X"

Two lemmas that may prove useful for the next proof.

lemma *small_steps_SeqE*: "$(c1 ;; c2, s) \rightarrow* (SKIP, s')$
$\Longrightarrow \exists t. (c1, s) \rightarrow* (SKIP, t) \wedge (c2, t) \rightarrow* (SKIP, s')$"
by (*induction* "$c1 ;; c2$" s *SKIP* s' *arbitrary*: $c1$ $c2$ *rule*: *star_induct*)
(*blast intro*: *star.step*)

lemma *small_steps_SeqI*: "$\llbracket(c1,\ s) \to* (SKIP,\ s'); (c2,\ s') \to* (SKIP,\ t)\rrbracket$
$\Longrightarrow (c1\ ;;\ c2,\ s) \to* (SKIP,\ t)$"
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**
"$c1 \sim_s c2 \longleftrightarrow (\forall s\ t.\ (c1,\ s) \to* (SKIP,\ t) \longleftrightarrow (c2,\ s) \to* (SKIP,\ t))$"

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 "$c1 \sim_s c2 \longleftrightarrow c1 \sim c2$" **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*: "$vars\ c1 \cap vars\ c2 = \{\}$"
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*: "$(c1\ ;;\ c2,\ s) \to* (SKIP,\ t)$" **and** *vars*: "$vars\ c1 \cap vars\ c2 = \{\}$"
have "$(c2\ ;;\ c1,\ s) \to* (SKIP,\ t)$"
} **with** *vars* **show** *?thesis* **unfolding** *equiv_com_def* **by** (*metis Int_commute*)
qed

Homework 8.1 Idempotence of Dead Varibale Elimination

Submission until Tuesday, December 17, 2013, 10:00am.

Dead variable elimination (*bury*) is not idempotent: multiple passes may reduce a command further and further. Give an example where $bury\ (bury\ c\ X)\ X \neq bury\ c\ X$. Hint: a sequence of two assignments.

Now define the textually identical function *bury* in the context of true liveness analysis (theory *Live_True*).

fun *bury* :: *"com ⇒ vname set ⇒ com"* **where**
"bury SKIP X = SKIP" |
"bury (x ::= a) X = (if x ∈ X then x ::= a else SKIP)" |
"bury (c_1;; c_2) X = (bury c_1 (L c_2 X);; bury c_2 X)" |
"bury (IF b THEN c_1 ELSE c_2) X = IF b THEN bury c_1 X ELSE bury c_2 X" |
"bury (WHILE b DO c) X = WHILE b DO bury c (L (WHILE b DO c) X)"

The aim of this homework is to prove that this version of *bury* is idempotent. This will involve reasoning about *lfp*. In particular we will need that *lfp* is the least pre-fixpoint. This is expressed by two lemmas from the library:

lfp_unfold: $mono\ ?f \Longrightarrow lfp\ ?f = ?f\ (lfp\ ?f)$
lfp_lowerbound: $?f\ ?A \leq ?A \Longrightarrow lfp\ ?f \leq ?A$

Prove the following lemma for showing that two fixpoints are the same, where *mono_def*: $mono\ ?f = (\forall x\ y.\ x \leq y \longrightarrow ?f\ x \leq ?f\ y)$.

lemma *lfp_eq*: "$[\![$ $mono\ f$; $mono\ g$; $lfp\ f \subseteq U$; $lfp\ g \subseteq U$;
!!X. $X \subseteq U \Longrightarrow f\ X = g\ X$ $]\!] \Longrightarrow lfp\ f = lfp\ g$"

It says that if we have an upper bound U for the *lfp* of both f and g, and f and g behave the same below U, then they have the same *lfp*.

The following two tweaks improve proof automation:

lemmas [*simp*] = *L.simps(5)*
lemmas *L_mono2* = *L_mono*[*unfolded mono_def*]

To show that *bury* is idempotent we need a lemma:

lemma *L_bury*[*simp*]: "$X \subseteq Y \Longrightarrow L\ (bury\ c\ Y)\ X = L\ c\ X$"
proof(*induction c arbitrary*: *X Y*)

The proof is straightforward except for the case *WHILE b DO c*. The definition of L in this case means that we have to show an equality of two *lfp*s. Lemma $[\![mono\ ?f;\ mono\ ?g;\ lfp\ ?f \subseteq ?U;\ lfp\ ?g \subseteq ?U;\ \bigwedge X.\ X \subseteq ?U \Longrightarrow ?f\ X = ?g\ X]\!] \Longrightarrow lfp\ ?f = lfp\ ?g$ comes to the rescue. We recommend the upper bound $lfp\ (\lambda Z.\ vars\ b \cup Y \cup L\ c\ Z)$. One of the two upper bound assumptions of lemma $[\![mono\ ?f;\ mono\ ?g;\ lfp\ ?f \subseteq ?U;\ lfp\ ?g \subseteq ?U;\ \bigwedge X.\ X \subseteq ?U \Longrightarrow ?f\ X = ?g\ X]\!] \Longrightarrow lfp\ ?f = lfp\ ?g$ can be proved by showing that U is a pre-fixpoint of f or g (see lemma *lfp_lowerbound*).

Now we can prove idempotence of *bury*, again by induction on c, but this time even the *While* case should be easy.

lemma *bury_bury*: "$X \subseteq Y \Longrightarrow bury\ (bury\ c\ Y)\ X = bury\ c\ X$"

Idempotence is a corollary:

corollary "$bury\ (bury\ c\ X)\ X = bury\ c\ X$"

Homework 8.2 Independence, in Parallel Programs

Submission until Tuesday, December 17, 2013, 10:00am. 5 bonus points.

Extend the while language with a parallel operator $\|$, such that $c1 \| c2$ executes commands $c1$ and $c2$ in parallel, and define a small step semantics.

Show that, for your parallel language, you have $vars\ c1 \cap vars\ c2 = \{\} \Longrightarrow (c1\ \|\ c2) \sim_s (c1\ ;;\ c2)$, i.e., sequential composition can be transformed to parallel execution if it works on different variables.

To solve this exercise, use the template from the webpage, which provides a sample solution from exercise 8.1 adapted to the parallel commands.

end