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

WS 2015/16
8. 12. 2015

Semantics of Programming Languages

Exercise Sheet 9

Exercise 9.1 Available Expressions

Regard the following function AA, which computes the *available assignments* of a command. An available assignment is a pair of a variable and an expression such that the variable holds the value of the expression in the current state. The function $AA\ c\ A$ computes the available assignments after executing command c, assuming that A is the set of available assignments for the initial state.

Note that available assignments can be used for program optimization, by avoiding recomputation of expressions whose value is already available in some variable.

fun AA :: *"com ⇒ (vname × aexp) set ⇒ (vname × aexp) set"* **where**
"AA SKIP A = A" |
"AA (x ::= a) A = (if x ∈ vars a then {} else {(x, a)})
∪ {(x′, a′). (x′, a′) ∈ A ∧ x ∉ {x′} ∪ vars a′}" |
"AA (c_1;; c_2) A = (AA c_2 ∘ AA c_1) A" |
"AA (IF b THEN c_1 ELSE c_2) A = AA c_1 A ∩ AA c_2 A" |
"AA (WHILE b DO c) A = A ∩ AA c A"

Show that available assignment analysis is a gen/kill analysis, i.e., define two functions *gen* and *kill* such that
$AA\ c\ A = (A \cup gen\ c) - kill\ c.$

Note that the above characterization differs from the one that you have seen on the slides, which is $(A - kill\ c) \cup gen\ c$. However, the same properties (monotonicity, etc.) can be derived using either version.

fun *gen* :: *"com ⇒ (vname × aexp) set"*
and *"kill"* :: *"com ⇒ (vname × aexp) set"*

lemma *AA_gen_kill*: *"AA c A = (A ∪ gen c) − kill c"*

Hint: Defining *gen* and *kill* functions for available assignments will require *mutual recursion*, i.e., *gen* must make recursive calls to *kill*, and *kill* must also make recursive calls to *gen*. The **and**-syntax in the function declaration allows you to define both functions simultaneously with mutual recursion. After the **where** keyword, list all the equations for both functions, separated by | as usual.

Now show that the analysis is sound:

theorem *AA_sound*:
"$(c,\ s) \Rightarrow s' \Longrightarrow \forall (x,\ a) \in AA\ \ c\ \{\}.\ s'\ x = aval\ a\ s'$"

Hint: You will have to generalize the theorem for the induction to go through.

Homework 9.1 Idempotence of Dead Varibale Elimination

Submission until Tuesday, December 15, 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 9.2 Dead Variables

Submission until Tuesday, Dec 15, 10:00am. **5 bonus points, quite easy!**

A variable is dead at a program point, if on all executions from that program point, it is not read before it is written.

Write a function that propagates sets of dead variables backwards through a command:

fun D :: "$com \Rightarrow vname\ set \Rightarrow vname\ set$"

Show the following correspondence between dead and live variable analysis:

lemma "$D\ c\ X = -\ L\ c\ (-X)$"

Note, $-\ X \equiv UNIV\ -\ X$ is set complement.