Technische Universität München
Fakultät für Informatik
Dr. Peter Lammich
Simon Wimmer

WS 2018/19
22.01.2019

Semantics of Programming Languages

Exercise Sheet 13

Exercise 13.1 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 (c₁;; c₂) A = (AA c₂ ∘ AA c₁) A" |
  "AA (IF b THEN c₁ ELSE c₂) A = AA c₁ A ∩ AA c₂ 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:

using idempotence!!

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 13.1 While-Loops with Independent Condition

Submission until Tuesday, January 29, 2019, 10:00am.

The following is the (slightly modified) text of an old exam exercise. In a real exam, we would ask you to solve the exercise on paper. For now, you should still use Isabelle. We expect you to write a detailed Isar proof. You also need to prove the auxiliary lemmas and proof steps that the students were allowed to skip in the exercise below.

Consider the big-semantics for IMP. We will denote the set of variable of a command c by *vars c*.

Question 1 Show:

theorem *ex1*:
  shows "$(WHILE\ b\ DO\ c,\ s) \Rightarrow t \Longrightarrow vars\ c \cap vars\ b = \{\} \Longrightarrow \neg\ bval\ b\ s$"

Hint: You may use the following fact:

$$vars\ c \cap vars\ b = \{\} \Longrightarrow (c,\ s) \Rightarrow t \Longrightarrow bval\ b\ s \longleftrightarrow bval\ b\ t$$

Question 2 Define a function *no* such that *no c* holds if and only if c contains no while loops. Show:

theorem *ex2*:
  "$no\ c \Longrightarrow \forall\ s.\ \exists\ t.\ (c,\ s) \Rightarrow t$"

Hint: You may skip the cases for *SKIP* and assignment when performing an induction.

Homework 13.2 Copy Propagation

Submission until Tuesday, January 29, 2019, 10:00am.

In this exercise, we are going to extend the available assignments analysis from the tutorial to a *copy propagation* program transformation. The idea is to eliminate variables that contain copied values as far as possible.

To do so, we will use an adapted version of the available expressions analysis from the tutorial. Modify the analysis such that it only considers assignments of the form $x ::= y$.

fun AA :: *"com ⇒ (vname × vname) set ⇒ (vname × vname) set"* **where**

Analogously to the tutorial, show that the analysis is a gen/kill analysis.

fun *gen* :: *"com ⇒ (vname × vname) set"*
and *kill* :: *"com ⇒ (vname × vname) set"*
theorem *AA_gen_kill*: *"AA c A = (A ∪ gen c) − kill c"*

Prove the following auxiliary properties of AA (using *AA_gen_kill*).

lemma *AA_distr*: *"AA c (A1 ∩ A2) = AA c A1 ∩ AA c A2"*
lemma *AA_idemp*: *"AA c (AA c A) = AA c A"*

Show soundness of AA:

theorem *AA_sound*:
  "(c,s) ⇒ s′ ⟹ ∀ (x,y) ∈ A. s x = s y ⟹ ∀ (x,y) ∈ AA c A. s′ x = s′ y"

We are now ready to define copy propagation. We use a substitution function on *aexp* to eliminate copied variables:

fun *subst* :: *"(vname ⇒ vname) ⇒ aexp ⇒ aexp"* **where**
  "subst f (N n) = N n" |
  "subst f (V y) = V (f y)" |
  "subst f (Plus a1 a2) = Plus (subst f a1) (subst f a2)"

Prove the following substitution lemma:

lemma *subst_lemma*: *"aval (subst σ a) s = aval a (λx. s (σ x))"*

We now turn a set of available assignments into a substitution:

definition
  "to_map A x = (if ∃ y. (x, y) ∈ A then SOME y. (x, y) ∈ A else x)"

Complete the following definition of the copy propagation. The second parameter gives the set of assignments that are available initially. Each of your cases should perform non-trivial but sound copy propagation. You do not need to alter Boolean expressions.

fun *CP* **where**
  "CP SKIP A = SKIP"
| *"CP (x ::= a) A = (x ::= subst (to_map A) a)"*

Our goal is to prove soundness of the program transformation: $c \sim CP\ c\ \{\}$. You may use the following lemma:

lemma *to_map_eq*:
  assumes *"∀(x,y)∈A. s x = s y"*
  shows *"(λx. s (to_map A x)) = s"*
  using *assms* **unfolding** *to_map_def*
  by (*auto del*: *ext intro*!: *ext*) (*metis* (*mono_tags*, *lifting*) *old.prod.case someI_ex*)

As for the liveness analysis, we split the proof into two directions. Prove the first direction:

theorem *CP_correct1*:
  assumes "$(c,\ s) \Rightarrow t$" "$\forall (x,y){\in}A.\ s\ x = s\ y$"
  shows "$(CP\ c\ A,\ s) \Rightarrow t$"

Hint: The theorems *assign_simp* and *fun_upd_def* may also be helpful.

Bonus (up to four points): Prove the other direction.

theorem *CP_correct2*:
  assumes "$(CP\ c\ A,\ s) \Rightarrow t$" "$\forall (x,y){\in}A.\ s\ x = s\ y$"
  shows "$(c,\ s) \Rightarrow t$"

Now the final theorem follows trivially:

corollary *CP_correct*:
  "$c \sim CP\ c\ \{\}$"
  apply (*rule equivI*)
   apply (*erule CP_correct1*, *simp*)
  apply (*erule CP_correct2*, *simp*)
  done