**Technische Universität München**
**Institut für Informatik**
**Prof. Tobias Nipkow, Ph.D.**
**Lars Hupel**

**WS 2017/18**
**12.12.2017**

# 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.

## General homework instructions

All proofs in the homework must be carried out in Isar style.

## Homework 9.1 Definite initialization analysis

*Submission until Tuesday, December 19, 10:00am.*

Define the definite initialization analysis as two recursive functions *ivars* and *ok* such that *ivars* computes the set of definitely initialized variables and *ok* checks that only initialized variables are accessed.

**fun** *ivars* :: "$com \Rightarrow vname\ set$"
**fun** *ok* :: "$vname\ set \Rightarrow com \Rightarrow bool$"
**lemma** "$D\ A\ c\ A' \Longrightarrow A' = A \cup ivars\ c \wedge ok\ A\ c$"
**lemma** "$ok\ A\ c \Longrightarrow D\ A\ c\ (A \cup ivars\ c)$"

## Homework 9.2 Dependencies

*Submission until Tuesday, December 19, 10:00am.*

*Hint:* Use the template file; it contains further instructions and definitions.

The task is to define a dependency analysis between variables. We say that variable $x$ depends on $y$ after command $c$ if the value of $y$ at the beginning of the execution of $c$ may influence the value of $x$ at the end of the execution.

For example, consider the program $y ::= 0;\ IF\ x \le 2\ THEN\ y ::= x\ ELSE\ z ::= 0$.

Here, the variable $x$ depends only on itself, since it is never assigned.

The variable $y$ clearly depends on $x$. It does not depend on itself, since it is initially assigned a constant value, hence the original value is irrelevant.

The variable $z$ depends on itself, since it may keep its value, but it also depends on $x$, since the assignment to it occurs under a conditional depending on $x$.

In the program $WHILE\ b\ DO\ (x ::= y;\ y ::= z)$ the variable $x$ depends on both $y$ and $z$ (the value of $z$ reaches $x$ in the second iteration of the loop).

1. Define an inductive relation *influences* :: $name \Rightarrow com \Rightarrow name \Rightarrow bool$ which specifies a dependency analysis.
2. Prove its soundness w.r.t. to the big-step semantics.