Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch, Kevin Kappelmann

WS 23/24
12.12.2023

Semantics of Programming Languages

Exercise Sheet 8

Exercise 8.1 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.

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 {(x, a)} else {})
   ∪ {(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:

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 8 Be Original!

Submission until Monday, Dec 25, 23:59pm.

Think up a nice topic to formalize yourself! It can be from any area of mathematics, computer science, etc., but should contain some interesting proof(s) – mere definitions or implementations are not interesting.

Creativity is encouraged and will be graded, but keep in mind that formalizations can often be more difficult than anticipated. Set yourself realistic goals! You are also welcome to discuss your project with us beforehand.

Comment your formalization well such that we can see what it is intended to do.

Incomplete or unfinished formalizations are welcome and will be graded (but clean them up so it is obvious what is there and what is missing).

The project will run until the winter holiday. This week, it makes up for half of the homework, and next week the full homework –

in total, this exercise will be worth 15 points, plus bonus points for nice submissions.

Homework 8 Denoting the ol' Switcheroo

Submission until Monday, Dec 18, 23:59pm.

Note: For this homework, wait until after the Thursday lecture on denotational semantics.

Define a denotational semantics for our IMP with the nondeterministic *SWITCH CASES* construct. Define the semantics in a functional way:

- one equation per construct, no *if* / *case*
- in the *SWITCH CASES* equation, use list functions such as *map* and *set*
- it is not allowed to define any auxiliary function (W is already given)

fun D :: "*com* ⇒ *com_den*"

Now adapt the correctness proofs w.r.t. to the big-step semantics – do not use *smt*, *metis*, *meson*, *moura*, or *argo* (i.e., most sledgehammer-generated proofs).

lemma *D_if_big_step*: "$(c,s) \Rightarrow t \Longrightarrow (s,t) \in D(c)$"
lemma *Big_step_if_D*: "$(s,t) \in D(c) \Longrightarrow (s,t) \in$ *Big_step c*"

theorem *denotational_is_big_step*:
"$(s,t) \in D(c)\ =\ ((c,s) \Rightarrow t)$"

by (*blast intro*: *D_if_big_step Big_step_if_D*[*simplified*])