Technische Universität München
Fakultät für Informatik
Prof. Tobias Nipkow, Ph.D.
Simon Wimmer

WS 2019/20
02.12.2019

Semantics of Programming Languages

Exercise Sheet 07

Exercise 7.1 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 $AA :: \text{“}com \Rightarrow (vname \times aexp)\ set \Rightarrow (vname \times aexp)\ set\text{”}$ **where**
$\text{“}AA\ SKIP\ A = A\text{”}$ |
$\text{“}AA\ (x ::= a)\ A = (if\ x \in vars\ a\ then\ \{\}\ else\ \{(x, a)\})$
$\cup \{(x', a').\ (x', a') \in A \wedge x \notin \{x'\} \cup vars\ a'\}\text{”}$ |
$\text{“}AA\ (c_1;;\ c_2)\ A = (AA\ c_2 \circ AA\ c_1)\ A\text{”}$ |
$\text{“}AA\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ A = AA\ c_1\ A \cap AA\ c_2\ A\text{”}$ |
$\text{“}AA\ (WHILE\ b\ DO\ c)\ A = A \cap AA\ c\ A\text{”}$

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 $gen :: \text{“}com \Rightarrow (vname \times aexp)\ set\text{”}$
and $kill :: \text{“}com \Rightarrow (vname \times aexp)\ set\text{”}$
lemma AA_gen_kill: $\text{“}AA\ c\ A = (A \cup gen\ c) - kill\ c\text{”}$

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.

Exercise 7.2 Security type system: bottom-up with subsumption

Recall security type systems for information flow control from the lecture. Such a type systems can either be defined in a top-down or in a bottom-up manner. Independently of this choice, the type system may or may not contain a subsumption rule (also called anti-monotonicity in the lecture). The lecture discussed already all but one combination: a bottom-up type system with subsumption.

1. Define a bottom-up security type system for information flow control with subsumption rule (see below, add the subsumption rule).
2. Prove the equivalence of the newly introduced bottom-up type system with the bottom-up type system without subsumption rule from the lecture.

inductive *sec_type2'* :: "*com* ⇒ *level* ⇒ *bool*" ("(⊢' _ : _)" [0,0] 50) **where**
Skip2': "⊢' *SKIP* : *l*" |
Assign2': "*sec x* ≥ *sec a* ⟹ ⊢' *x* ::= *a* : *sec x*" |
Semi2': "⟦ ⊢' c_1 : *l*; ⊢' c_2 : *l* ⟧ ⟹ ⊢' c_1 ;; c_2 : *l*" |
If2': "⟦ *sec b* ≤ *l*; ⊢' c_1 : *l*; ⊢' c_2 : *l* ⟧ ⟹ ⊢' *IF b THEN* c_1 *ELSE* c_2 : *l*" |
While2': "⟦ *sec b* ≤ *l*; ⊢' *c* : *l* ⟧ ⟹ ⊢' *WHILE b DO c* : *l*"

Homework 7.1 Security Typing

Submission until Monday, Dec 9, 10:00am.

In this homework, you should define a function that erases confidential ("private") parts of a command:

fun *erase* :: "*level* ⇒ *com* ⇒ *com*" **where**

Function *erase l* should replace all assignments to variables with security level ≥ *l* by *SKIP*. It should also erase certain *IF*s and *WHILE*s, depending on the security level of the Boolean condition. Now show that *c* and *erase l c* behave the same on the variables up to level *l*:

theorem *erase_correct*:
"⟦ (*c*,*s*) ⇒ *s'*; (*erase l c*,*t*) ⇒ *t'*; *0* ⊢ *c*; *s* = *t* (< *l*) ⟧
⟹ *s'* = *t'* (< *l*)"

This lemma looks remarkably like the noninterference lemma in *HOL−IMP.Sec_Typing* (although ≤ has been replaced by <). You may want to start with that proof and modify

it where needed. A lot of local modifications will be necessary, but the structure should remain the same. You may also need one or two simple additional lemmas (for example $\ldots \Longrightarrow aval\ a\ s_1 = aval\ a\ s_2$), but nothing major.

In the theorem above we assumed that both $(c,\ s)$ and $(erase\ l\ c,\ t)$ terminate. How about the following two properties:

lemma "$[\![\ (c,s) \Rightarrow s';\ \ 0 \vdash c;\ \ s = t\ (< l)\ ]\!]$
$\Longrightarrow \exists\, t'.\ (erase\ l\ c,t) \Rightarrow t' \wedge s' = t'\ (< l)$"
lemma "$[\![\ (erase\ l\ c,s) \Rightarrow s';\ \ 0 \vdash c;\ \ s = t\ (< l)\ ]\!] \Longrightarrow \exists\, t'.\ (c,t) \Rightarrow t'$"

Give an informal justification or a counterexample for each property!

Homework 7.2 Definite Initialization

Submission until Monday, Dec 9, 10:00am.

A well-initialized command only depends on the variables that are already initialized. That is, executability and the values of the definitely initialized variables after executing the command only depend on the values of the already initialized variables before the command.

Prove the following lemma, which formalizes the proposition above wrt. the standard big-step semantics.

theorem *well_initialized_commands*:
assumes "*D A c B*"
assumes "*s1 = s2 on A*"
assumes "$(c,s1) \Rightarrow s1'$"
shows "$\exists\, s2'.\ (c,s2) \Rightarrow s2' \wedge s1' = s2'$ *on B*"