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

WS 2022/23
6.12.2022

Semantics of Programming Languages

Exercise Sheet 7

Exercise 7.1 Security type system: bottom-up with subsumption

Recall security type systems for information flow control from the lecture. Such a type system 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.

- Define a bottom-up security type system for information flow control with subsumption rule (see below, add the subsumption rule).
- 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"* |
Seq2′: *"⟦ ⊢′ c₁ : l; ⊢′ c₂ : l ⟧ ⟹ ⊢′ c₁ ;; c₂ : l"* |
If2′: *"⟦ sec b ≤ l; ⊢′ c₁ : l; ⊢′ c₂ : l ⟧ ⟹ ⊢′ IF b THEN c₁ ELSE c₂ : l"* |
While2′: *"⟦ sec b ≤ l; ⊢′ c : l ⟧ ⟹ ⊢′ WHILE b DO c : l"*

lemma *"⊢ c : l ⟹ ⊢′ c : l"*

lemma *"⊢′ c : l ⟹ ∃ l′ ≥ l. ⊢ c : l′"*

Exercise 7.2 Available Expressions

Regard the 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 $AA\ c\ A$ below 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.

Why does the assignment case need to check if x is in a?

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"*

Now show that the analysis is sound:

theorem *AA_sound*:
  "(c, s) ⇒ s' ⟹ ∀(x, a) ∈ AA c {}. s' x = aval a s'"

Hint: You will have to generalize the theorem for the induction to go through. You may assume idempotency of *AA* in the proof:

lemma *AA_idem*: *"AA c (AA c A) = AA c A"*

Of course we still need to prove the the idempotency lemma – but you will find that a straightforward proof is quite difficult. An easier solution is to find an equivalent formulation with two functions where the first one specifies which assignments *AA* adds to *A* (*gen*), and the second one which it removes (*kill*). Those functions need to be mutually recursive; you can add the equations for both below.

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

Examples:

lemma *"gen (''x''::=N 5;; ''x''::= N 6) = {(''x'',N 6)}"*
  by *simp*

lemma *"(''x'', N 6) ∉ kill (''x''::=N 5;; ''x''::= N 6)"*
  by *simp*

For this formulation, the idempotency lemma should be straightforward, as should the proof that they are equal:

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

Note that in the lecture, *gen*/*kill* will be defined slightly differently.

Homework 7.1 Security Typing

Submission until Monday, Dec 12, 23:59pm.

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

fun *erase* :: *"level ⇒ com ⇒ com"*

Function *erase l* should replace all assignments to variables with security level $\geq l$ 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*:
"⟦ $(c,s) \Rightarrow s'$; $(\mathit{erase}\ l\ c,t) \Rightarrow t'$; $0 \vdash c$; $s = t\ (< l)$ ⟧
$\Longrightarrow s' = t'\ (< l)$"

This lemma looks remarkably like the noninterference lemma in *HOL−IMP.Sec_Typing* (although $\leq$ 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 $\ldots \Longrightarrow \mathit{aval}\ a\ s_1 = \mathit{aval}\ a\ s_2$), but nothing major.

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

prop "⟦ $(c,s) \Rightarrow s'$; $0 \vdash c$; $s = t\ (< l)$ ⟧
$\Longrightarrow \exists t'.\ (\mathit{erase}\ l\ c,t) \Rightarrow t' \wedge s' = t'\ (< l)$"
prop "⟦ $(\mathit{erase}\ l\ c,s) \Rightarrow s'$; $0 \vdash c$; $s = t\ (< l)$ ⟧ $\Longrightarrow \exists t'.\ (c,t) \Rightarrow t'$"

Give an informal justification or a counterexample for each property!

Homework 7.2 Definite Initialization

Submission until Monday, Dec 12, 23:59pm.

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 "$D\ A\ c\ B$"
and "$s = s'$ *on* A"
and "$(c,s) \Rightarrow t$"
and "$(c,s') \Rightarrow t'$"
shows "$t{=}t'$ *on* B"