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

WS 2021/22
7.12.2021

Semantics of Programming Lectures

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 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.

- 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': "$\vdash'$ *SKIP* : l" |
Assign2': "$sec\ x \geq sec\ a \Longrightarrow \vdash'\ x ::= a : sec\ x$" |
Seq2': "$[\![\ \vdash'\ c_1 : l;\ \ \vdash'\ c_2 : l\ ]\!] \Longrightarrow \vdash'\ c_1\ ;;\ c_2 : l$" |
If2': "$[\![\ sec\ b \leq l;\ \ \vdash'\ c_1 : l;\ \ \vdash'\ c_2 : l\ ]\!] \Longrightarrow \vdash'\ IF\ b\ THEN\ c_1\ ELSE\ c_2 : l$" |
While2': "$[\![\ sec\ b \leq l;\ \ \vdash'\ c : l\ ]\!] \Longrightarrow \vdash'\ WHILE\ b\ DO\ c : l$"
lemma "$\vdash\ c : l \Longrightarrow \vdash'\ c : l$"
lemma "$\vdash'\ c : l \Longrightarrow \exists\, l' \geq l.\ \vdash\ c : l'$"

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**

$\text{“}AA\ SKIP\ A = A\text{”}$
$|\ \text{“}AA\ (x ::= a)\ A = (if\ x \notin vars\ a\ then\ \{(x,\ a)\}\ else\ \{\})$
$\quad \cup \{(x',\ a').\ (x',\ a') \in A \wedge x \notin \{x'\} \cup vars\ a'\}\text{”}$
$|\ \text{“}AA\ (c_1;;\ c_2)\ A = (AA\ c_2 \circ AA\ c_1)\ A\text{”}$
$|\ \text{“}AA\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ A = AA\ c_1\ A \cap AA\ c_2\ A\text{”}$
$|\ \text{“}AA\ (WHILE\ b\ DO\ c)\ A = A \cap AA\ c\ A\text{”}$

Now show that the analysis is sound:

theorem *AA_sound*:
$\text{“}(c,\ s) \Rightarrow s' \Longrightarrow \forall (x,\ a) \in AA\ c\ \{\}.\ s'\ x = aval\ a\ s'\text{”}$

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*: $\text{“}AA\ c\ (AA\ c\ A) = AA\ c\ A\text{”}$

Now prove the idempotency lemma.

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* :: $\text{“}com \Rightarrow (vname \times aexp)\ set\text{”}$ **and** *kill* :: $\text{“}com \Rightarrow (vname \times aexp)\ set\text{”}$

Examples:

lemma $\text{“}gen\ (''x''{::=}N\ 5;;\ ''x''{::=}\ N\ 6) = \{(''x'',N\ 6)\}\text{”}$
by *simp*

lemma $\text{“}(''x'',\ N\ 6) \notin kill\ (''x''{::=}N\ 5;;\ ''x''{::=}\ N\ 6)\text{”}$
by *simp*

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

lemma *AA_gen_kill*: $\text{“}AA\ c\ A = (A \cup gen\ c) - kill\ c\text{”}$

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

Homework 7.1 Terminating While Loops

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

The objective of this homework is to identify while loops of the form *while* $x{<}n$ *do* c, such that x is a variable, n is a constant, and the execution of command c is guaranteed to increment x.

Your first task is to write a function that checks whether a command is guaranteed to increment a variable. Note: This predicate can only be an approximation. You are not required to use information of conditions, nor to track other variables than the regarded one. You need only consider arithmetic expressions of form $y + k$ (for variables y and constants k).

Hint: An auxiliary function *invar* that checks whether a command is guaranteed to preserve the value of a variable may be helpful:

fun *invar* :: *"vname ⇒ com ⇒ bool"*
fun *incr* :: *"vname ⇒ com ⇒ bool"*

Some tests for the approximation:

value *"incr ''x'' (''x'' ::= Plus (V ''x'') (N 0);; ''x'' ::= Plus (V ''x'') (N 2)) = True"*
value *"incr ''y'' (*
WHILE Less (V ''y'') (N 2)
DO
''y'' ::= Plus (V ''y'') (N 1);;
''x''::=Plus (V ''x'') (N (−1))
) = False"

Prove that your approximation is correct:

lemma *incr_less*: *"(c,s) ⇒ t ⟹ incr x c ⟹ s x < t x"*

Use your approximation to write a termination checker, that accepts programs where all the while-loops are of the form described above.

fun *terminates* :: *"com ⇒ bool"*

Prove that your termination checker only accepts terminating programs:

$terminates\ c \Longrightarrow \exists\, t.\ (c,\ s) \Rightarrow t$

For that, you will need the the crucial auxiliary lemma, namely that whenever c always terminates and increments x, then also the while-loop *while* $(x<k)$ c always terminates.

You may use the induction rule *less_induct* for an inductive argument over the difference of x and k:

$(\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ a$

The rule works natural numbers; you can use *nat* to convert an int to nat.

lemma *term_w*:
assumes *step*: *"⋀s. ∃t. (c,s) ⇒ t"*
and *incr*: *"incr x c"*
shows *"∃t. (WHILE Less (V x) (N k) DO c, s) ⇒ t"*

Finally, prove:

theorem *term_big_step*: *"terminates c ⟹ ∃ t. (c,s) ⇒ t"*

Homework 7.2 A Typed Language

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

We unify boolean expressions *bexp* and arithmetic expressions *aexp* into one expressions language *exp*. We also define a datatype *val* to represent either integers or booleans. We then give a type system and small semantics. Your task is to show preservation and progress of the type system.

Preparation 1: We define unified values and expressions:

datatype $val = Iv\ int \mid Bv\ bool$

datatype $exp = N\ int \mid V\ (char\ list) \mid Plus\ exp\ exp \mid Bc\ bool \mid Not\ exp \mid And\ exp\ exp \mid Less\ exp\ exp$

Evaluation is now defined as an inductive predicate only working when the types of the values are correct, i.e.:

$eval\ (N\ i)\ s\ (Iv\ i)$

$eval\ (V\ x)\ s\ (s\ x)$

$[\![eval\ a_1\ s\ (Iv\ i_1);\ eval\ a_2\ s\ (Iv\ i_2)]\!] \Longrightarrow eval\ (Plus\ a_1\ a_2)\ s\ (Iv\ (i_1 + i_2))$

$eval\ (Bc\ v)\ s\ (Bv\ v)$

$eval\ b\ s\ (Bv\ bv) \Longrightarrow eval\ (Not\ b)\ s\ (Bv\ (\neg\ bv))$

$[\![eval\ b_1\ s\ (Bv\ bv_1);\ eval\ b_2\ s\ (Bv\ bv_2)]\!] \Longrightarrow eval\ (And\ b_1\ b_2)\ s\ (Bv\ (bv_1 \wedge bv_2))$

$[\![eval\ a_1\ s\ (Iv\ i_1);\ eval\ a_2\ s\ (Iv\ i_2)]\!] \Longrightarrow eval\ (Less\ a_1\ a_2)\ s\ (Bv\ (i_1 < i_2))$

Preparation 2: The small-step semantics are as before, we just replaced *aval* and *bval* with *eval*:

$eval\ a\ s\ v \Longrightarrow (x ::= a,\ s) \to (SKIP,\ s(x := v))$

$eval\ b\ s\ (Bv\ True) \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \to (c_1,\ s)$

$eval\ b\ s\ (Bv\ False) \Longrightarrow (IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \to (c_2,\ s)$

Preparation 3: We introduce the type system:

datatype $ty = Ity \mid Bty$

$\Gamma \vdash N\ i : Ity$

$\Gamma \vdash V\ x : \Gamma\ x$

$[\![\Gamma \vdash a_1 : Ity;\ \Gamma \vdash a_2 : Ity]\!] \Longrightarrow \Gamma \vdash Plus\ a_1\ a_2 : Ity$

$\Gamma \vdash Bc\ v : Bty$

$\Gamma \vdash b : Bty \Longrightarrow \Gamma \vdash Not\ b : Bty$

$[\![\Gamma \vdash b_1 : Bty;\ \Gamma \vdash b_2 : Bty]\!] \Longrightarrow \Gamma \vdash And\ b_1\ b_2 : Bty$

$[\![\Gamma \vdash a_1 : Ity;\ \Gamma \vdash a_2 : Ity]\!] \Longrightarrow \Gamma \vdash Less\ a_1\ a_2 : Bty$

$\Gamma \vdash \mathit{SKIP}$
$\Gamma \vdash a : \Gamma\ x \Longrightarrow \Gamma \vdash x ::= a$
$[\![\Gamma \vdash c_1;\ \Gamma \vdash c_2]\!] \Longrightarrow \Gamma \vdash c_1;;\ c_2$
$[\![\Gamma \vdash b : \mathit{Bty};\ \Gamma \vdash c_1;\ \Gamma \vdash c_2]\!] \Longrightarrow \Gamma \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2$
$[\![\Gamma \vdash b : \mathit{Bty};\ \Gamma \vdash c]\!] \Longrightarrow \Gamma \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c$

We define a state typing *styping* to describe the type context of a state:
$\mathit{type}\ (\mathit{Iv}\ i) = \mathit{Ity}$
$\mathit{type}\ (\mathit{Bv}\ r) = \mathit{Bty}$

$(\Gamma \vdash s) = (\forall x.\ \mathit{type}\ (s\ x) = \Gamma\ x)$

Task: Show progress and then soundness of the type system:

theorem *progress*: "$\Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow c \neq \mathit{SKIP} \Longrightarrow \exists\, cs'.\ (c,s) \rightarrow cs'$"
theorem *type_sound*:
"$(c,s) \rightarrow\!*\ (c',s') \Longrightarrow \Gamma \vdash c \Longrightarrow \Gamma \vdash s \Longrightarrow c' \neq \mathit{SKIP} \Longrightarrow \exists\, cs''.\ (c',s') \rightarrow cs''$"

Hint: For most of the proof work, you should be able to closely follow the proofs in the original IMP theory.