Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D., Dr. Peter Lammich
Johannes Hölzl

WS 2014/15
2. 12. 2014

Semantics of Programming Languages

Exercise Sheet 8

Exercise 8.1 Security type system: bottom-up with subsumption

Use the template file `ex08_tmpl.thy`.

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.
2. Prove the equivalence of the newly introduced bottom-up type system with the bottom-up type system without subsumption rule from the lecture.

Exercise 8.2 Definite Initialization Analysis

Use the template file `ex08_tmpl.thy`.

In the lecture, you have seen a definite initialization analysis that was based on the big-step semantics. Definite initialization analysis can also be based on a small-step semantics. Furthermore, the ternary predicate D from the lecture can be split into two parts: a function $AA :: com \Rightarrow name\ set$ ("assigned after") which collects the names of all variables assigned by a command and a binary predicate $D :: name\ set \Rightarrow com \Rightarrow bool$ which checks that a command accesses only previously assigned variables. Conceptually, the ternary predicate from the lecture (call it D_{lec}) and the two-step approach should relate by the equivalence $D\ V\ c \longleftrightarrow D_{lec}\ V\ c\ (V \cup AA\ c)$

1. Study the already defined small-step semantics for definite analysis.
2. Define the function AA which computes the set of variables assigned after execution of a command. Furthermore, define the predicate D which checks if a command accesses only assigned variables, assuming the variables in the argument set are already assigned.
3. Prove progress and preservation of D with respect to the small-step semantics, and conclude soundness of D. You may use (and then need to prove) the lemmas *D_incr* and *D_mono*.

Homework 8.1 A Typed Language

Submission until Tuesday, December 9, 2014, 10:00am.

Use the template file `hw08_tmpl.thy`.

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, i.e. replace all `oops` by valid proofs.

Preparation 1: We define unified values and expressions:

datatype *val* = *Iv int* | *Bv bool*
datatype *exp* =
  N int | *V vname* | *Plus exp exp* | *Bc bool* | *Not exp* | *And exp exp* | *Less exp exp*

Evaluation is now defined as inductive predicate only working when the types of the values are correct:

inductive *eval* :: *"exp ⇒ state ⇒ val ⇒ bool"* **where**
"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) ⟹ eval (Plus a_1 a_2) s (Iv (i_1 + i_2))" |
"eval (Bc v) s (Bv v)" |
"eval b s (Bv bv) ⟹ eval (Not b) s (Bv (¬ bv))" |
"eval b_1 s (Bv bv_1) ⟹ eval b_2 s (Bv bv_2) ⟹ eval (And b_1 b_2) s (Bv (bv_1 ∧ bv_2))" |
"eval a_1 s (Iv i_1) ⟹ eval a_2 s (Iv i_2) ⟹ 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*.

inductive
  small_step :: *"(com × state) ⇒ (com × state) ⇒ bool"* (**infix** *"→" 55*)
where
Assign: *"eval a s v ⟹ (x ::= a, s) → (SKIP, s(x := v))"* |
IfTrue: *"eval b s (Bv True) ⟹ (IF b THEN c_1 ELSE c_2,s) → (c_1,s)"* |
IfFalse: *"eval b s (Bv False) ⟹ (IF b THEN c_1 ELSE c_2,s) → (c_2,s)"* |

...

Preparation 3: We introduce the type system.

datatype *ty* = *Ity* | *Bty*

type_synonym *tyenv* = *"vname ⇒ ty"*

inductive *etyping* :: *"tyenv ⇒ exp ⇒ ty ⇒ bool"*
 (*"(1_/ ⊢/ (_ :/ _))"*)
where
"Γ ⊢ N i : Ity" |
"Γ ⊢ V x : Γ x" |
"Γ ⊢ a_1 : Ity ⟹ Γ ⊢ a_2 : Ity ⟹ Γ ⊢ Plus a_1 a_2 : Ity" |

"Γ ⊢ *Bc v* : *Bty*" |
"Γ ⊢ *b* : *Bty* ⟹ Γ ⊢ *Not b* : *Bty*" |
"Γ ⊢ b_1 : *Bty* ⟹ Γ ⊢ b_2 : *Bty* ⟹ Γ ⊢ *And* b_1 b_2 : *Bty*" |
"Γ ⊢ a_1 : *Ity* ⟹ Γ ⊢ a_2 : *Ity* ⟹ Γ ⊢ *Less* a_1 a_2 : *Bty*"

inductive *ctyping* :: "*tyenv* ⇒ *com* ⇒ *bool*" (**infix** "⊢" *50*) **where**
"Γ ⊢ *SKIP*" |
"Γ ⊢ *a* : Γ *x* ⟹ Γ ⊢ *x* ::= *a*" |
"Γ ⊢ c_1 ⟹ Γ ⊢ c_2 ⟹ Γ ⊢ c_1;;c_2" |
"Γ ⊢ *b* : *Bty* ⟹ Γ ⊢ c_1 ⟹ Γ ⊢ c_2 ⟹ Γ ⊢ *IF b THEN* c_1 *ELSE* c_2" |
"Γ ⊢ *b* : *Bty* ⟹ Γ ⊢ *c* ⟹ Γ ⊢ *WHILE b DO c*"

We define a state typing *styping* to describe the type context of a state.

fun *type* :: "*val* ⇒ *ty*" **where**
"*type* (*Iv i*) = *Ity*" |
"*type* (*Bv r*) = *Bty*"
definition *styping* :: "*tyenv* ⇒ *state* ⇒ *bool*" (**infix** "⊢" *50*) **where**
"Γ ⊢ *s* ⟷ (∀ *x*. *type* (*s x*) = Γ *x*)"

Task 1: Show preservation and progress on expressions:

lemma *epreservation*: "Γ ⊢ *a* : τ ⟹ *eval a s v* ⟹ Γ ⊢ *s* ⟹ *type v* = τ"
lemma *eprogress*: "Γ ⊢ *a* : τ ⟹ Γ ⊢ *s* ⟹ ∃ *v*. *eval a s v*"

Task 2: Show progress and preservation on commands:

theorem *progress*: "Γ ⊢ *c* ⟹ Γ ⊢ *s* ⟹ *c* ≠ *SKIP* ⟹ ∃ *cs'*. (*c*,*s*) → *cs'*"
theorem *styping_preservation*: "(*c*,*s*) → (*c'*,*s'*) ⟹ Γ ⊢ *c* ⟹ Γ ⊢ *s* ⟹ Γ ⊢ *s'*"
theorem *ctyping_preservation*: "(*c*,*s*) → (*c'*,*s'*) ⟹ Γ ⊢ *c* ⟹ Γ ⊢ *c'*"
theorem *type_sound*:
"(*c*,*s*) →* (*c'*,*s'*) ⟹ Γ ⊢ *c* ⟹ Γ ⊢ *s* ⟹ *c'* ≠ *SKIP* ⟹ ∃ *cs''*. (*c'*,*s'*) → *cs''*"

Homework 8.2 Terminating While Loops

Submission until Tuesday, December 9, 2014, 10:00am. 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.

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

fun *invar* :: "*vname* ⇒ *com* ⇒ *bool*"
— True if command does not change value of variable
fun *incr* :: "*vname* ⇒ *com* ⇒ *bool*"
— True if command increments value of variable

Some tests for your approximation: These should all evaluate to True

abbreviation *"x ≡ ''x''"* **abbreviation** *"y ≡ ''y''"*
abbreviation *"L ≡ (WHILE (Less (V y) (N 2)) DO y ::= Plus (V y) (N 1))"*
abbreviation *"I n ≡ x ::= Plus (V x) (N n)"*
value *"incr x (I 1;; I 2)"* **value** *"incr x (I 1;; L)"*
value *"incr x (L ;; I 1)"* **value** *"incr x (IF (Less (V y) (N 1)) THEN I 1 ELSE I 2)"*

You may return False on the following

value *"incr x (I −1;; I 2)"* **value** *"incr x (x::= V x;; I 1)"*

Prove that your approximation is correct

lemma *incr*: *"(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. You may formulate your termination checker as function or as inductive predicate. Both has its advantages and disadvantages:

function It is hard to forget some case, but you need to use the induction rule generated by the function, which does not have nice case names.

inductive The induction rule has nice case names, however, it is easy to forget some cases, and define a predicate that excludes too many terminating programs.

inductive *"term"* :: *"com ⇒ bool"* **where**

Prove that your termination checker only accepts terminating programs. Hint: The following induction rule may be helpful:

lemma *int_less_induct*:
assumes *"⋀i. i ≥ k ⟹ P i"* **and** *"⋀i. i < k ⟹ (⋀j. i < j ⟹ P j) ⟹ P i"*
shows *"P(i::int)"* — Proof provided in auxiliary file

Then prove 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 *int_less_induct* for an inductive argument over the difference of x and k.

lemma *term_w*: **assumes** *"⋀s. EX t. (c,s) ⇒ t" "incr x c"*
shows *"EX t. (WHILE Less (V x) (N k) DO c, s) ⇒ t"*
proof(*induction "s x" arbitrary*: *s rule*: *int_less_induct*[*of k*])

Finally, prove:

theorem *"term c ⟹ EX t. (c,s) ⇒ t"*