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

WS 2020/21
14.12.2020

Semantics of Programming Languages

Exercise Sheet 7

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.

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.

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'* :: "$com \Rightarrow level \Rightarrow bool$" ("$(\vdash''\ \_ : \_)$" $[0,0]$ 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 "$\vdash' c : l \Longrightarrow \exists l' \geq l.\ \vdash c : l'$"

theory *hw07* **imports** "*HOL−IMP.Sec_Type_Expr*" "*HOL−IMP.Def_Init_Small*" **begin**

Homework 7.1 Security type systems: bottom-up vs. top-down

Submission until Sunday, Dec 20, 23:59.

Prove the equivalence of the bottom-up system ($\vdash \_ : \_$) and the top-down system ($\_ \vdash \_$) without subsumption rule. Carry out a direct correspondence proof in both directions without using the $\vdash'$ system.

lemma "$\vdash c : l \Longrightarrow l \vdash c$"
lemma *top_down_impl_bottom_up*: "$l \vdash c \Longrightarrow \exists\ l' \geq l.\ \vdash c : l'$"

Homework 7.2 Definite Initialization Analysis

Submission until Sunday, Dec 20, 23:59.

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 AV :: *com* ⇒ *name set* ("assigned variables") which collects the names of all variables assigned by a command and a binary predicate D :: *name set* ⇒ *com* ⇒ *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 $D\ V\ c \longleftrightarrow D_{lec}\ V\ c\ (V \cup AV\ c)$

hide_const *D*

Define the functions AV, and D (which checks a command accesses only assigned variables, assuming the variables in the argument set are already assigned).

```
fun AV :: "com ⇒ vname set"
fun D :: "vname set ⇒ com ⇒ bool"
```

Progress is already proven for you as the proof is quite similar to D_{lec}. If the proof doesn't work, check your definitions!

```
theorem D_progress:
  assumes "c ≠ SKIP"
  shows "D (dom s) c ⟹ ∃ cs'. (c,s) → cs'"
  using assms
proof (induction c arbitrary: s)
  case Assign thus ?case by auto (metis aval_Some small_step.Assign)
next
  case (If b c1 c2)
  then obtain bv where "bval b s = Some bv" by (auto dest!: bval_Some)
  then show ?case
    by(cases bv) (auto intro: small_step.IfTrue small_step.IfFalse)
qed (fastforce intro: small_step.intros)+
```

Now we want to prove preservation of D with respect to the small-step semantics, and from progress and preservation conclude soundness of D. You may use (and then need to prove) the lemmas *D_incr* and *D_mono*.

Proofs like this can often seem magical when using the full proof automation of Isabelle, which makes them very hard to read. Hence, your proofs for this exercise may *not* contain any of the advanced solvers like fastforce, smt, blast, etc.

This means that only the following proof methods should be used:

- *cases/induction*
- *rule*
- ./..

- *simp*/*simp_all*
- *auto* without arguments, if the context (*this*) contains at most a single fact

Of course all of the isar syntax (fix, obtain, ...), forward proofs, instantiation, local lemmas, etc. are still allowed.
Give the proofs in Isar, not apply-style!

lemma *D_incr*: "$(c,s) \to (c',s') \Longrightarrow dom\ s \cup AV\ c \subseteq dom\ s' \cup AV\ c'$"

lemma *D_mono*: "$A \subseteq A' \Longrightarrow D\ A\ c \Longrightarrow D\ A'\ c$"

theorem *D_preservation*: "$(c,s) \to (c',s') \Longrightarrow D\ (dom\ s)\ c \Longrightarrow D\ (dom\ s')\ c'$"

theorem *D_sound*: "$(c,s) \to* (c',s') \Longrightarrow c' \neq SKIP \Longrightarrow D\ (dom\ s)\ c \Longrightarrow \exists\, cs''.\ (c',s') \to cs''$"

Homework 7.3 Be Original! (Topic Selection)

Submission until Sunday, Dec 20, 23:59. Think up a nice topic to formalize yourself, for example

- Prove some interesting result about algorithms/graphs/automata/formal language theory
- Formalize some results from mathematics
- Find interesting modifications of IMP material and prove interesting properties about them
- ...

This week, you should select the topic and start to formalize some concepts from it. Be creative! You will have time until after the winter break (Jan 10) to finish the project. *In total, this exercise will be worth 15 points, plus bonus points for nice submissions.*
You are welcome to discuss your plans with the tutor (via e-mail) before starting. This is, however, not a necessity by any means.
For the actual formalization, note the following:

- you should set yourself a time limit before you start
- also incomplete/unfinished formalizations are welcome and will be graded
- comment your formalization well, such that we can see what it does/is intended to do