Concrete Semantics with Isabelle/HOL

Exercise Sheet 9

Exercise 9.1 Definite Initialization Analysis

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 \( AA : \text{com} \Rightarrow \text{name set} \) (“assigned after”) which collects the names of all variables assigned by a command and a binary predicate \( D : \text{name set} \Rightarrow \text{com} \Rightarrow \text{bool} \) which checks that a command accesses only previously assigned variables. Conceptually, the ternary predicate from the lecture (\( \text{Def}\_\text{Init}.D \)) and the two-step approach should relate by the equivalence \( D A c \iff \text{Def}\_\text{Init}.D A c (A \cup AA c) \).

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 \( D_{\text{incr}} \) and \( D_{\text{mono}} \).

\[
\text{inductive} \\
\text{small\_step} :: \text{“com} \times \text{state} \Rightarrow \text{com} \times \text{state} \Rightarrow \text{bool}” \text{ (infix “\(\rightarrow\)” 55)}
\]

\[
\text{lemmas} \text{ small\_step\_induct} = \text{small\_step\_induct}[\text{split\_format(complete)}]
\]

\[
\text{inductive} \\
\text{small\_steps} :: \text{“com} \ast \text{state} \Rightarrow \text{com} \ast \text{state} \Rightarrow \text{bool}” \text{ (infix “\(\rightarrow\)∗” 55)}
\]

\[
\text{lemmas} \text{ small\_steps\_induct} = \text{small\_steps\_induct}[\text{split\_format(complete)}]
\]

\[
\text{fun} AA :: \text{“com} \Rightarrow \text{name set}”
\]

\[
\text{fun} D :: \text{“vname set} \Rightarrow \text{com} \Rightarrow \text{bool}”
\]

\[
\text{theorem} \text{ progress}: “D (\text{dom } s) c \iff c \neq \text{SKIP} \Rightarrow \exists c’. (c, s) \rightarrow c’”
\]

\[
\text{lemma} D_{\text{incr}}: “(c, s) \rightarrow (c’, s’) \Rightarrow \text{dom } s \cup AA c \subseteq \text{dom } s’ \cup AA c’”
\]

\[
\text{lemma} D_{\text{mono}}: “A \subseteq A’ \Rightarrow D A c \Rightarrow D A’ c”
\]

\[
\text{theorem} D_{\text{preservation}}: “(c, s) \rightarrow (c’, s’) \Rightarrow D (\text{dom } s) c \Rightarrow D (\text{dom } s’) c’”
\]
\textbf{Exercise 9.2} \ Available Definitions

An \textit{available definition} analysis determines which previous assignments $x := a$ are valid equalities $x = a$ at later program points. For example, after $x := y + 1$ the equality $x = y + 1$ is available, but after $x := y + 1; y := 2$ the equality $x = y + 1$ is no longer available. The motivation for the analysis is that if $x = a$ is available before $v := a$ then $v := a$ can be replaced by $v := x$.

Define an available definitions analysis as a gen/kill analysis, for suitably defined $\textit{gen\_ad}$ and $\textit{kill\_ad}$ (which may need to be mutually recursive).

\begin{verbatim}
fun
  gen\_ad :: "com ⇒ (vname * aexp) set" and
  kill\_ad :: "com ⇒ (vname * aexp) set"

definition AD :: "(vname * aexp) set ⇒ com ⇒ (vname * aexp) set" where
  "AD A c = (A − kill\_ad c) ∪ gen\_ad c"
\end{verbatim}

A call $AD A c$ should compute the available definitions after the execution of $c$ assuming that the definitions in $A$ are available before the execution of $c$.

\textbf{Remark:} The gen/kill framework is introduced in the book on a so-called \textit{backward analysis}: In $L c X$, the input $X$ is the set of live variables \textit{after} the execution of $c$, and $L c X$ computes the live variables \textit{before}. In contrast, the available definition analysis will take a set of available definitions before executing $c$ and return the available definitions after. The different orientation means that you have to be especially careful when implementing the sequential composition case.

Prove correctness of the analysis:

\begin{verbatim}
theorem "(c, s) ⇒ s′ ⇒ ∀(x, a) ∈ A. s x = AExp.aval a s ⇒
  ∀(x, a) ∈ AD A c. s′ x = AExp.aval a s"
\end{verbatim}

\textbf{Warning:} Be careful with your import files. The definite analysis theory files ($\textit{Def\_Init\_}$) redefine semantic functions (e.g., $\textit{aval}$). Either remove these imports or qualify the symbols you need (e.g., $\textit{AExp.aval}$).
Homework 9.1  Commuting Commands (20 points)

Submission until Monday 06.07.2015 10:00 AM. Please send your submissions to jasmin.blanchette@mpi-inf.mpg.de and mathias.fleury@mpi-inf.mpg.de with “[CONCRETE9]” in the subject of your email. If your name is Prenome Cognome, please call your theory file Cognome.Prenome.thy.

Commands generally do not commute with respect to sequential composition, meaning that \(c1 ;; c2\) and \(c2 ;; c1\) have a different semantic. However, if the variables of \(c1\) and \(c2\) are disjoint, the two commands commute.

**Step A** (2 points): Write a brief informal step-by-step sketch of the proof.

**Step B** (9 points): Prove basic lemmas that will be useful for our main theorem. Below are some suggestions. You might want to attack Step C first and find out which lemmas will be useful.

Hint: The \(f = g\ on\ A\) operator is defined in theory Vars and is described in Section 10.3.2 of the book. The complement of a set \(A\) is \(\text{UNIV} - A\) or \(- A\).

**lemma eq_on_antimono[intro]**: “\(A \subseteq B \implies f = g\ on\ B \implies f = g\ on\ A\)”

**lemma big_step_changes_only_vars**: “\((c, s) \Rightarrow t \implies t = s\ on\ (- vars\ c)\)”

**lemma change_notin_vars_v1**: 
assumes “\(s = sa\ on\ vars\ c\) and “\((c, s) \Rightarrow t\)”
shows “\((c, sa) \Rightarrow (\lambda x. (if x \in\ vars\ c\ then\ t\ else\ sa)\ x)\)”

**lemma change_notin_vars_v2**: 
assumes “\(s = sa\ on\ vars\ c\) and “\((c, s) \Rightarrow t\)”
shows “\(\exists ta.\ t = ta\ on\ vars\ c\ \land\ (c, sa) \Rightarrow ta\)”

**lemma change_notin_vars_v3**: 
assumes “\(s = sa\ on\ vars\ c\) and “\((c, s) \Rightarrow t\) and “\((c, sa) \Rightarrow ta\)”
shows “\(t = ta\ on\ vars\ c\)”

**Step C** (9 points): Prove the commutativity theorem.

**theorem disjoint_imp_Seq_commute**: 
assumes “\(\text{vars}\ c1 \cap\ \text{vars}\ c2 = \{\}\)” and 
step: “\((\text{Seq}\ c1\ c2, s) \Rightarrow t\)”
shows “\((\text{Seq}\ c2\ c1, s) \Rightarrow t\)”