DD2457 Program Semantics and Analysis

Examination problems
with solution sketches
3 June 2015, 14:00 - 19:00

Dilian Gurov
KTH CSC
tel: 08-790 8198

1 Level E

1. Consider the *Natural semantics* of the **While** language. The rules of the semantics do not impose a particular order of execution. However, we discussed in class that the language is deterministic, and that a strategy can be imposed on applying the rules that guarantees that for any given initial configuration $\langle S, s\rangle$ the exploration will terminate whenever there exists a derivation. $\boxed{4p}$

   (a) Suggest a procedure (or function) that executes **While** programs in Natural semantics from a specified initial state, yielding the final state if there is one. Provide the pseudo-code of the procedure in a suitable notation.

   Solution: Here is a possible solution, implemented as a function:

```
fun exec(S, s) =
   let fun update(s, x, v) = ... end;
   let fun aeval(a, s) =
          case a of
                 n : n;
                 x : s(x);
             a1+a2 : aeval(a1, s) + aeval(a2, s);
             a1-a2 : aeval(a1, s) - aeval(a2, s);
             a1*a2 : aeval(a1, s) * aeval(a2, s);
          end;
       end;
   let fun beval(b, s) = ... end;
   in case S of
                    x:=a : update(s, x, aeval(a, s));
                    skip : s;
                   S1;S2 : let s’ = exec(S1, s) in exec(S2, s’);
       if b then S1 else S2 : beval(b, s) = tt ? exec(S1, s) ! exec(S2, s);
               while b do S : beval(b, s) = tt
                                ? let s’ = exec(S, s) in exec(while b do S, s’);
                                ! s;
      end;
end;
```

   (b) Explain briefly the workings of your procedure (or function), and argue why it yields the correct final state whenever it exists.

   Solution: The evaluation of arithmetic and Boolean expressions follows the semantic definitions of $\mathcal{A}$ and $\mathcal{B}$, which we have shown in class to be deterministic. The rules for statements are uniquely applicable depending on the type of statement and its potential Boolean guard, except for the choice of state s' in rules $[\mathsf{comp}_{\mathsf{ns}}]$ and $[\mathsf{while}^{\mathsf{tt}}_{\mathsf{ns}}]$. Again, we have shown in class that execution of **While** programs in Natural semantics is deterministic, and so we are safe to compute s' by executing S_1, respectively S, from s.

2. Consider the following (not particularly useful) **While** program: $\boxed{4\text{p}}$

```
$x := 17;$
**while** $0 \leq y$ **do**
    **if** $x \leq y$ **then**
        $y := y - x;$
        **while** $0 \leq x$ **do** $x := x - y$
    **else**
        $x := x - 1;$
```

Your task is to construct for this program a *test suite*, i.e., a set of tests, each being represented simply by an initial state. The *coverage criterion* for the suite is to traverse every possible simple path of the program, i.e., no edge of the control flow graph needs to be visited more than once along the same path. Apply *symbolic execution* (see handouts) to construct such a test suite. Illustrate and explain your construction step-by-step.

Solution: As we are only interested in simple paths, one possibility is to re-write all **while** statements to **if** statements (for simplicity I will omit the **else** branches), and to add labels as a preparatory step:

```
$l_0 : x := 17;$
$l_1 :$ **if** $0 \leq y$ **then**
    $l_2 :$ **if** $x \leq y$ **then**
        $l_3 : y := y - x;$
        $l_4 :$ **if** $0 \leq x$ **then** $l_5 : x := x - y$
    **else**
        $l_6 : x := x - 1;$
$l_F :$
```

The paths in this modified program (corresponding to the simple paths of the original one), each corresponding path condition obtained by symbolic execution as explained in class, and a possible corresponding satisfying assignment (i.e., state, if there is one), are summarized by the table:

Path	*Path Condition*	*State Identifier*	*State*
$l_0l_1l_2l_3l_4l_5l_F$	$0 \leq y_0 \land 17 \leq y_0 \land 0 \leq 17$	s_1	$[x \mapsto 0, y \mapsto 17]$
$l_0l_1l_2l_3l_4l_F$	$0 \leq y_0 \land 17 \leq y_0 \land \neg(0 \leq 17)$	$-$	$-$
$l_0l_1l_2l_6l_F$	$0 \leq y_0 \land \neg(17 \leq y_0)$	s_3	$[x \mapsto 0, y \mapsto 0]$
$l_0l_1l_F$	$\neg(0 \leq y_0)$	s_4	$[x \mapsto 0, y \mapsto -1]$

The second path is infeasible, since its path condition is unsatisfiable, and we thus obtain the test suite $T = \{s_1, s_3, s_4\}$.

2 Level C

For grade D you need to have passed level E and obtained 5 (out of 12) points from this section. For passing level C you need 8 points from this section.

Consider the extension of **While** with the **thread** S **end** statement discussed in class and in Assignment 2. Extend the *Abstract Machine* of Chapter 4 of the course book to handle this extension in a way that you still can argue for correctness of the implementation of **While**. That is:

1. Propose an operational semantics for the extended language that is suitable as a *specification* for the implementation. $\boxed{3\text{p}}$

   Solution: The first subtle point of the task is that, as discussed in class and in the course book, Natural semantics is inadequate to capture the interleaving of multi-threaded execution. We have therefore to pick a SOS-style semantics. Here is a set of possible rules, following a suggestion presented in class, based on configurations over multi-sets of statement sequences (see also the suggested solution to problem E1 from the exam from 21 May 2013):

$$\begin{array}{rll}
[\mathsf{skip_{MT}}] & \langle(\mathbf{skip} : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle\gamma + \mathcal{M}, s\rangle & \\
[\mathsf{ass_{MT}}] & \langle(x := a : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle\gamma + \mathcal{M}, s[x \mapsto \mathcal{A}[\![a]\!]\,(s)]\rangle & \\
[\mathsf{thread_{MT}}] & \langle(\mathbf{thread}\ S\ \mathbf{end} : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle S + \gamma + \mathcal{M}, s\rangle & \\
[\mathsf{comp_{MT}}] & \langle(S_1; S_2 : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle(S_1 : S_2 : \gamma) + \mathcal{M}, s\rangle & \\
[\mathsf{if^{tt}_{MT}}] & \langle(\mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle(S_1 : \gamma) + \mathcal{M}, s\rangle & \text{if } \mathcal{B}[\![b]\!]\,(s) = \mathbf{tt} \\
[\mathsf{if^{ff}_{MT}}] & \langle(\mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle(S_2 : \gamma) + \mathcal{M}, s\rangle & \text{if } \mathcal{B}[\![b]\!]\,(s) = \mathbf{ff} \\
[\mathsf{while^{tt}_{MT}}] & \langle(\mathbf{while}\ b\ \mathbf{do}\ S : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle(S : \mathbf{while}\ b\ \mathbf{do}\ S : \gamma) + \mathcal{M}, s\rangle & \text{if } \mathcal{B}[\![b]\!]\,(s) = \mathbf{tt} \\
[\mathsf{while^{ff}_{MT}}] & \langle(\mathbf{while}\ b\ \mathbf{do}\ S : \gamma) + \mathcal{M}, s\rangle \Rightarrow \langle\gamma + \mathcal{M}, s\rangle & \text{if } \mathcal{B}[\![b]\!]\,(s) = \mathbf{ff}
\end{array}$$

   It is against this semantics as a specification of **While** that the correctness of the implementation will be argued.

2. In this operational semantics, show an execution of the program: $\boxed{2\text{p}}$

$$x := 0;\ \mathbf{while\ true\ do}\ (x := x + 1;\ \mathbf{thread}\ x := x - 1\ \mathbf{end})$$

   from an arbitrary state s up to a repeating configuration.

   Solution: Let W abbreviate **while true do** ($x := x + 1$; **thread** $x := x - 1$ **end**).

$$\begin{array}{lll}
 & \langle x := 0;\ W, s\rangle & \\
\Rightarrow & \langle W, s[x \mapsto 0]\rangle & [\mathsf{ass_{MT}}] \\
\Rightarrow & \langle(x := x + 1;\ \mathbf{thread}\ x := x - 1\ \mathbf{end}); W, s[x \mapsto 0]\rangle & [\mathsf{while^{tt}_{MT}}] \\
\Rightarrow & \langle(x := x + 1;\ \mathbf{thread}\ x := x - 1\ \mathbf{end}) : W, s[x \mapsto 0]\rangle & [\mathsf{comp_{MT}}] \\
\Rightarrow & \langle x := x + 1 : \mathbf{thread}\ x := x - 1\ \mathbf{end} : W, s[x \mapsto 0]\rangle & [\mathsf{comp_{MT}}] \\
\Rightarrow & \langle \mathbf{thread}\ x := x - 1\ \mathbf{end} : W, s[x \mapsto 1]\rangle & [\mathsf{ass_{MT}}] \\
\Rightarrow & \langle(x := x - 1) + W, s[x \mapsto 1]\rangle & [\mathsf{thread_{MT}}] \\
\Rightarrow & \langle W, s[x \mapsto 0]\rangle & [\mathsf{ass_{MT}}]
\end{array}$$

   and we reached a repeating configuration.

3. Extend suitably the abstract machine language **AM** and the translation of **While** into **AM**. $\boxed{\text{2p}}$

   Solution: We could extend **AM** with the instruction SPAWN(c), and the translation of statements with the clause:

$$\mathcal{CS}[\![\textbf{thread } S \textbf{ end}]\!] \overset{\mathsf{def}}{=} \text{SPAWN}(\mathcal{CS}[\![\text{S}]\!])$$

4. Describe the necessary extensions of the (operational semantics of the) abstract machine, so that you can still argue for correctness of the implementation (this you will do in part A below). $\boxed{\text{3p}}$

   Solution: We extend abstract machine configurations to $\langle C, e, s\rangle$, where C is a multi-set of code (instruction sequences), and add the operational semantics rule:

$$\langle \text{SPAWN}(c) : c' + C, e, s\rangle \triangleright \langle c + c' + C, e, s\rangle$$

   However, here comes the next subtle point: The SOS specification of **While** treats the evaluation of arithmetic and Boolean expressions *atomically*, and thus the current operational semantic rules of the abstract machine do not guarantee correctness of the implementation.

   One way to achieve such atomicity on the abstract machine level could be to restrict the rules so that, whenever the evaluation stack is non-empty, only the code (thread) that last made the evaluation stack non-empty is allowed to progress, until the stack becomes empty again.

5. Translate the program from item 2 above to **AM** and execute the resulting code from an arbitrary state s up to a repeating configuration, matching the execution from item 2. $\boxed{\text{2p}}$

   Solution: The translation of the program proceeds as follows:

$$\begin{aligned}
& \mathcal{CS}[\![x := 0;\ \textbf{while true do}\ (x := x+1;\ \textbf{thread}\ x := x-1\ \textbf{end})]\!] \\
= \ & \mathcal{CS}[\![x := 0]\!] : \mathcal{CS}[\![\textbf{while true do}\ (x := x+1;\ \textbf{thread}\ x := x-1\ \textbf{end})]\!] \\
= \ & \text{PUSH}{-}0 : \text{STORE}{-}x : \text{LOOP}(\mathcal{CS}[\![\textbf{true}]\!], \mathcal{CS}[\![x := x+1;\ \textbf{thread}\ x := x-1\ \textbf{end}]\!]) \\
= \ & \text{PUSH}{-}0 : \text{STORE}{-}x : \text{LOOP}(\text{TRUE}, \mathcal{CS}[\![x := x+1]\!] : \mathcal{CS}[\![\textbf{thread}\ x := x-1\ \textbf{end}]\!]) \\
= \ & \text{PUSH}{-}0 : \text{STORE}{-}x : \text{LOOP}(\text{TRUE}, \text{PUSH}{-}1 : \text{FETCH}{-}x : \text{ADD} : \text{STORE}{-}x : \\
& \qquad\qquad\qquad\qquad \text{SPAWN}(\text{PUSH}{-}1 : \text{FETCH}{-}x : \text{SUB} : \text{STORE}{-}x))
\end{aligned}$$

   Let c abbreviate the code PUSH$-$1 : FETCH$-x$: ADD : STORE$-x$: SPAWN(PUSH$-$1 : FETCH$-x$: SUB : STORE$-x$). The execution that follows the one of the SOS above proceeds as follows, jumping over steps and presenting only the matching configurations:

$$\begin{aligned}
& \langle \text{PUSH}{-}0 : \text{STORE}{-}x : \text{LOOP}(\text{TRUE}, c), \varepsilon, s\rangle \\
\triangleright^+ \ & \langle \text{LOOP}(\text{TRUE}, c), \varepsilon, s[x \mapsto 0]\rangle \\
\triangleright^+ \ & \langle c : \text{LOOP}(\text{TRUE}, c), \varepsilon, s[x \mapsto 0]\rangle \\
\triangleright^+ \ & \langle \text{SPAWN}(\text{PUSH}{-}1 : \text{FETCH}{-}x : \text{SUB} : \text{STORE}{-}x) : \text{LOOP}(\text{TRUE}, c), \varepsilon, s[x \mapsto 1]\rangle \\
\triangleright \ & \langle \text{PUSH}{-}1 : \text{FETCH}{-}x : \text{SUB} : \text{STORE}{-}x \ + \ \text{LOOP}(\text{TRUE}, c), \varepsilon, s[x \mapsto 1]\rangle \\
\triangleright^+ \ & \langle \text{LOOP}(\text{TRUE}, c), \varepsilon, s[x \mapsto 0]\rangle
\end{aligned}$$

   and we reached the repeating configuration. Note that all transitions adhere to the restriction stated above that only the thread that last made the evaluation stack non-empty is allowed to progress.

3 Level A

For grade B you need to have passed level C and obtained 4 (out of 10) points from this section. For grade A you need 7 points from this section.

1. For the extension of **While** with multi-threading and its implementation developed in part C above, formalize the statement of *correctness of the implementation* (just the statement itself, not its proof). Argue as formally as you can that your correctness statement holds. $\boxed{\text{5p}}$

   Solution: First, we need to lift the translation of statements $\mathcal{CS}[\![-]\!]$ to multi-sets of statement sequences $\mathcal{M}$, in the obvious way; then the definition of $\mathcal{S}_{am}$ can stay as it is, referring to the lifted definition of the extended translation. The operational semantics of multi-threaded **While** induces a semantic function $\mathcal{S}_{mt}$ in the usual way. The correctness statement can then be presented as:

   For every statement S of multi-threaded **While**, we have:

   $$\mathcal{S}_{mt}\,[\![S]\!] = \mathcal{S}_{am}\,[\![S]\!]$$

   The proof of the statement can again be organized as four separate results, closely following the treatment in the book. Lemma 4.18 and Exercise 4.19 can stay unchanged. Lemma 4.21 can be reformulated as follows:

   $$\text{if } \langle S, s\rangle \Rightarrow \langle S' + \gamma, s'\rangle \text{ then } \langle \mathcal{CS}[\![S]\!], \varepsilon, s\rangle \rhd^{+} \langle \mathcal{CS}[\![S']\!] + \mathcal{CS}[\![\gamma]\!], \varepsilon, s'\rangle$$

   where S' accounts for a potentially spawned new statement, and γ for the statement sequence resulting from S within the current thread. Lemma 4.22 can be adapted accordingly.

   To then combine these lemmas to prove the correctness statement above one has to rely heavily on the restriction imposed on the execution of the abstract machine that only the thread that last made the evaluation stack non-empty is allowed to progress.

2. When we developed the *Denotational semantics* of **While**, we discussed a possible definition for **while** loops as follows: $\boxed{\text{5p}}$

   $$\begin{aligned}\mathcal{S}'_{ds}[\![\textbf{while } b \textbf{ do } S]\!]\,(s) = s' \quad \overset{\mathsf{def}}{\Longleftrightarrow} \quad & \exists s_0, \ldots, s_n \in \textbf{State}.\\ & \quad s = s_0 \wedge s' = s_n\\ & \wedge\ \forall i < n.\ (\mathcal{B}[\![b]\!]\,(s_i) = \textbf{tt} \wedge \mathcal{S}_{ds}[\![S]\!]\,(s_i) = s_{i+1})\\ & \wedge\ \mathcal{B}[\![b]\!]\,(s_n) = \textbf{ff}\end{aligned}$$

   Instead, we chose a definition based on fixed points (i.e., as a specific solution to a recursive semantic equation). Still, argue as formally as you can that the above definition is equivalent to the chosen definition. (Hint: Refer to the meaning of the i-th approximant of the semantic transformer corresponding to the loop.)

   Solution: We have to show that for all $s, s' \in \textbf{State}$:

   $$\mathcal{S}_{ds}[\![\textbf{while } b \textbf{ do } S]\!]\,(s) = s' \ \Leftrightarrow\ \mathcal{S}'_{ds}[\![\textbf{while } b \textbf{ do } S]\!]\,(s) = s'$$

   We have by definition:

   $$\mathcal{S}_{ds}[\![\textbf{while } b \textbf{ do } S]\!] = \cup_{i\geq 0}\ F^{i}_{b,S}(\emptyset)$$

   and so we have that $\mathcal{S}_{ds}[\![\textbf{while } b \textbf{ do } S]\!]\,(s) = s'$ if and only if there is $i \geq 0$ such that $F^{i}_{b,S}(\emptyset)(s) = s'$. In class we discussed that $F^{i}_{b,S}(\emptyset)(s) = s'$ holds when executing **while** b **do** S from s terminates in s' with executing the loop body S at most $i-1$ times. Let n be the smallest index such that $F^{n+1}_{b,S}(\emptyset)(s) = s'$. This means then that executing **while** b **do** S from s terminates in s' with executing the loop body S *exactly* n times. This corresponds precisely to the case formalized by $\mathcal{S}'_{ds}[\![\textbf{while } b \textbf{ do } S]\!]$.