4.8 Denotational Semantics

The text describes a denotational semantics for the simple imperative language given by the grammar

$$P ::= x := e \mid P_1; P_2 \mid \text{if } e \text{ then } P_1 \text{ else } P_2 \mid \text{while } e \text{ do } P.$$

Each program denotes a function from *states* to *states*, in which a *state* is a function from *variables* to *values*.

(a) Calculate the meaning $C[\![x := 1; x := x + 1;]\!](s_0)$ in approximately the same detail as that of the examples given in the text, where $s_0 = \lambda v \in \textit{variables}.\ 0$, giving every variable the value 0.

(b) Denotational semantics is sometimes used to justify ways of reasoning about programs. Write a few sentences, referring to your calculation in part (a), explaining why

$$C[\![x := 1; x := x + 1;]\!](s) = C[\![x := 2;]\!](s)$$

for every state s.

C[[x := e]](s) = modify(s,x, E[[e]](s))
C[[P_1;P_2]](s) = C[[P_2]](C[[P_1]](s))
C[[if e then P_1 else P_2]](s) = if E[[e]](s) then C[[P_1]](s) else C[[P_2]](s)
C[[while e do P]](s) = if not E[[e]](s) then s
else C[[while e do P]](C[[P]](s))

a) $C[\![x:=1;\ x:=x+1;]\!](s_0) = C[\![x:=x+1]\!](C[\![x:=1]\!](s_0))$

$s_1 = s_0[x \mapsto 1]$ ⟹ $= C[\![x:=x+1]\!](s_1) = s_1[x \mapsto E[\![x+1]\!](s_1)]$

$= s_1[x \mapsto (E[\![x]\!](s_1) + E[\![1]\!](s_1))] = s_1[x \mapsto 2]$ $\xrightarrow{s_2 = s_0[x \mapsto 2]}$ $= s_2$

b) We know that $C[\![x:=2]\!](s) = s[x \mapsto 2]$

and from (a) we figure out that:

$C[\![x:=1;\ x:=x+1]\!](s_0) = s_0[x \mapsto 2]$ for any s_0

so they are equivalent.

4.9 Semantics of Initialize-Before-Use

A nonstandard denotational semantics describing initialize-before-use analysis is presented in the text.

(a) What is the meaning of

$$\mathcal{C}[\![x := 0;\ y := 0;\ \text{if } x = y \text{ then } z := 0 \text{ else } w := 1\ ]\!](s_0)$$

in the state $s_0 = \lambda y \in \mathit{variables}.\mathit{uninit}$? Show how to calculate your answer.

(b) Calculate the meaning

$$\mathcal{C}[\![\text{if } x = y \text{ then } z := y \text{ else } z := w]\!](s)$$

in state s with $s(x) = \mathit{init}$, $s(y) = \mathit{init}$, and $s(v) = \mathit{uninit}$ or every other variable v.

$$\mathcal{C}[[x := e]](s) = \begin{cases} s[x \mapsto \mathit{init}], & \mathcal{E}[[e]](s) = Ok \\ \text{error}, & \text{otherwise} \end{cases}$$

$$\mathcal{C}[[P_1; P_2]](s) = \begin{cases} \text{error}, & \mathcal{C}[[P_1]](s) = \text{error} \\ \mathcal{C}[[P_2]](\mathcal{C}[[P_1]](s)), & \text{otherwise} \end{cases}$$

$$\mathcal{C}[[\text{if e then } P_1 \text{ else } P_2]](s) = \begin{cases} \text{error}, & \mathcal{E}[[e]](s) = \text{err} \\ & \text{or } \mathcal{C}[[P_1]](s) = \text{error} \\ & \text{or } \mathcal{C}[[P_2]](s) = \text{error} \\ \mathcal{C}[[P_1]](s) \oplus \mathcal{C}[[P_2]](s), & \text{otherwise} \end{cases}$$

a) $\mathcal{C}[\![x := 0 ;\ y := 0 ;\ \text{if } x = y \text{ then } z := 0 \text{ else } w := 1]\!](s_0)$

$s_0[x \mapsto init] = s_1$

$\mathcal{C}[\![y := 0 ;\ \text{if } x = y \text{ then } z := 0 \text{ else } w := 1]\!]\left(\mathcal{C}[\![x := 0]\!](s_0)\right)$

$s_1[y \mapsto init] = s_2$

$= \mathcal{C}[\![\text{if } x = y \text{ then } z := 0 \text{ else } w := 1]\!]\left(\mathcal{C}[\![y := 0]\!](s_1)\right)$

$= \mathcal{C}[\![z := 0]\!](s_2) \oplus \mathcal{C}[\![w := 1]\!](s_2) = s_2$

b) $\mathcal{C}[\![\text{if } x = y \text{ then } z := y \text{ else } z := w]\!](s) = \text{error}$

because $\mathcal{E}[\![w]\!](s) = \text{error}$ so $\mathcal{C}[\![z := w]\!](s) = \text{error}$

4.10

a) $\mathcal{V}[\![\text{if false then 0 else 1}]\!](\eta_0) = \text{integer}$

because $\mathcal{V}[\![\text{false}]\!](\eta_0) = \text{boolean}$ and $\mathcal{V}[\![0]\!](\eta_0) = \mathcal{V}[\![1]\!](\eta_0) = \text{integer}$

b) $\mathcal{V}[\![\text{let } x : \text{int} = e_1 \text{ in } (\text{if } e_2 \text{ then } e_1 \text{ else } x)]\!](\eta_0)$

$= \mathcal{V}[\![\text{if } e_2 \text{ then } e_1 \text{ else } x]\!](\overbrace{\eta_0[x \mapsto \text{integer}]}^{\eta_1})$

$= \text{integer}$ because $\mathcal{V}[\![e_2]\!](\eta_1) = \text{boolean}$, $\mathcal{V}[\![e_1]\!](\eta_1) = \mathcal{V}[\![x]\!](\eta_1) = \text{integer}$

c) $\mathcal{V}[\![let\ x{=}e_1\ in\ e_2]\!]\eta = \begin{cases} \mathcal{V}[\![e_2]\!](\eta[x \mapsto \sigma]) & \text{if } \mathcal{V}[\![e_1]\!] = \sigma \text{ and } \sigma = \text{integer or boolean} \\ type\_error & \text{otherwise} \end{cases}$

4.11

a) yes. The process computing body of g must wait for process evaluating e_2 (because of (if) part in the body)

b) In lazy evaluation the output is 1 without error

Using parallel evaluation, the process for evaluting e_2 faces error.

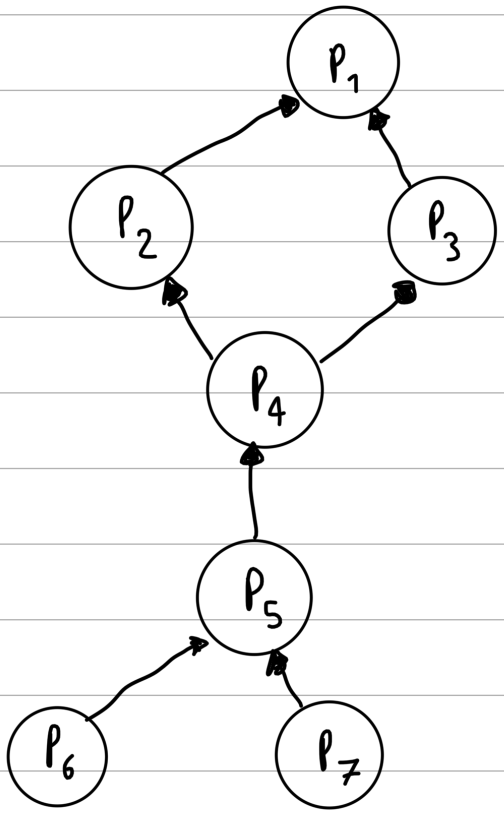
Depending on designer choice, you can get error or you can get the output 1 (because e_1 is 0)

c) As mentioned in b), you can simply report no error.

d) No, we can not. Because of race condition.

4.12

a) Dependency graphs of process:



Only P_2 and P_3 can be executed in parallel and both must wait for P_1. P_4 must wait for P_2 and P_3. P_5 must wait for P_4 and P_6, P_7 could be executed in parallel if the SA condition would be hold.

b) yes, we can (because x is assigned once (SA)). Also we can eliminate these calls from P_6 and P_7 and use the calculated values from P_2, P_3.

c) The parallel processes mentioned in (a) is correct because the calls to functions are call by value and don't have any side effects. But in (b) we use the fact that x is assigned once (SA).

d) No, this problem can be reduced to halting problem which is undecidable (Every non-trivial attribute of a program is undecidable). We can use some conservative tricks (like ⊕ for (if) branches) to detect single-assignment condition (We get soundness, not completeness)

e) yes, it does. Each variable is assigned once (the values does not change till the end of scope) so it passes declarative language test.