1 2014 Paper 6 Question 9

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2014p6q9.pdf

Consider the concurrent imperative language L with syntax and conventional operational semantics as below.

Say p, m has a data race if there is a sequence of transitions p, $m \xrightarrow{l_1} \ldots \xrightarrow{l_n} \xrightarrow{l} \xrightarrow{l'}$ were l and l' conflict: they are reads or writes t the same location, at least one is a write and they are by different threads.

(a) Give a p for which p, m_0 has a data race.

$$p = tid_1 : x := 0; \textbf{skip} \mid tid_2 : x := 1; \textbf{skip}$$

(b) A *vector clock* c is a function from thread ids to natural numbers, identifying the $c(tid)'$th transition of each thread tid. Modify the semantics above to add a vector clock c to each process thread ($tid_c : s$), each process label ($tid_c : l$), and each memory location (with each $m(x)$ now being a pair v_c of a value and vector clock). In your semantics each vector clock should be computed so as to record the latest transition number of all threads that have causally affected that point. Explain your semantics, perhaps with some simple examples.

$$\begin{aligned}
&\textit{statement}, s ::= \ldots \\
&\textit{expression}, e ::= \ldots \\
&\textit{process}, p ::= tid_c : s \mid p|p' \\
&\textit{label}, l ::= \tau \mid \mathbf{W}x = v \mid \mathbf{R}x = v \mid tid_c : \tau \mid tid_c : \mathbf{W}x = v \mid \ tid_c : \mathbf{R}x = v
\end{aligned}$$

$$\frac{}{x := v; s \xrightarrow{\mathbf{W}x=v} s}\ \text{WR} \qquad \frac{}{\textbf{let } r = x \textbf{ in } s \xrightarrow{\mathbf{R}x=v} \{v/r\}s}\ \text{RD}$$

$$\frac{}{\textbf{if } (v = v)\ s \textbf{ else } s' \xrightarrow{\tau} s}\ \text{IF1} \qquad \frac{v \neq v'}{\textbf{if } (v = v')\ s \textbf{ else } s' \xrightarrow{\tau} s'}\ \text{IF2}$$

$$\frac{v = [\![op]\!](v_1, \ldots, v_n)}{\textbf{let } r = op(v_1, \ldots, v_n) \textbf{ in } s \xrightarrow{\tau} \{v/r\}s}\ \text{OP}$$

$$\frac{s \xrightarrow{l} s'}{tid_c : s \xrightarrow{tid_c:l} tid_c : s'}\ \text{THREAD}$$

$$\frac{tid_c : s \xrightarrow{tid_c:\tau} tid_c : s' \qquad c' = c \oplus \{(tid, c(tid) + 1)\}}{tid_c : s, m \xrightarrow{tid_c:\tau} tid_{c'} : s', m}\ \text{STAU}$$

$$\frac{\begin{array}{c} tid_{c_0} : s \xrightarrow{tid_{c_0}:\mathbf{R}x=v} tid_{c_0} : s' \\ m(x) = (v, c_1) \\ c_2 = \{(t, i) | t \in (\text{dom}(c_0 \cup c_1) - \{tid\}) \wedge i = \max(c_0(t), c_1(t))\} \oplus \{(tid, \max(c_0(tid), c_1(tid)) + 1)\} \end{array}}{tid_{c_0} : s, m \xrightarrow{tid_{c_0}:\mathbf{R}x=v} tid_{c_2} : s', m}\ \text{SRD}$$

$$\frac{\begin{array}{c} tid_{c_0} : s \xrightarrow{tid_{c_0}:\mathbf{W}x=v} tid_{c_0} : s' \\ m(x) = (\_, c_1) \\ c_2 = \{(t, i) | t \in (\text{dom}(c_0 \cup c_1) - \{tid\}) \wedge i = \max(c_0(t), c_1(t))\} \oplus \{(tid, \max(c_0(tid), c_1(tid)) + 1)\} \end{array}}{tid_{c_0} : s, m \xrightarrow{tid_{c_0}:\mathbf{R}x=v} tid_{c_2} : s', m \oplus (x, (v, c_2))}\ \text{SWR}$$

$$\frac{p_1, m \xrightarrow{l} p_1', m'}{p_1|p_2, m \xrightarrow{l} p_1'|p_2, m'} \text{ PLEFT}$$

$$\frac{p_2, m \xrightarrow{l} p_2', m'}{p_1|p_2, m \xrightarrow{l} p_1|p_2', m'} \text{ PRIGHT}$$

The operations WR, RD, IF1, IF2 and OP are unchanged – these are thread local semantics and so are not affected by changes regarding thread IDs.

Labels are also added unconditionally to reductions.

The rules STAU, SRD and SWR allow updates to the global store and keep the vector clocks consistent.

The rules PLEFT and PRIGHT allow for interleaving of computation between processes.

The main difference from the previous semantics is that parallelism is at a process level rather than a thread level.

Consider the example below:

$$0_{[1,0]} : x := 1; \mathbf{skip} | 1_{[0,2]} : \mathbf{skip}, \{(x \mapsto (0, [0, 2]))\}$$

$$\dfrac{\dfrac{\dfrac{}{x := 1; \mathbf{skip} \xrightarrow{\mathbf{W}x=v} s}\text{ WR}}{0_{[1,0]} : x := 1; \mathbf{skip} \xrightarrow{0_{[1,0]}\mathbf{W}x=v} 0_{[1,0]} : \mathbf{skip}}\text{ THREAD} \qquad m(x) = (\_, [0, 2]) \qquad c_2 = [2, 2]}{0_{[1,0]} : x := 1; \mathbf{skip}, \{(x \mapsto (0, [0, 2]))\} \xrightarrow{0_{[1,0]}\mathbf{W}x=v} 0_{[2,2]} : \mathbf{skip}, \{(x \mapsto (1, [2, 2]))}\text{ SWR}}{0_{[1,0]} : x := 1; \mathbf{skip} | 1_{[0,2]} : \mathbf{skip}, \{(x \mapsto (0, [0, 2]))\} \xrightarrow{0_{[1,0]}\mathbf{W}x=v} 0_{[2,2]} : \mathbf{skip} | 1_{[0,2]} : \mathbf{skip}, \{(x \mapsto (1, [2, 2]))}\text{ PRIGHT}$$

(c) Suppose that $p, m \xrightarrow{l} \xrightarrow{l_1} \dots \xrightarrow{l_n} \xrightarrow{l'}$ in your vector clock semantics, where l and l' conflict but are separated by $l_1, \dots, l_n$. To implement a dynamic race detector, we would like to find conditions on $l_1, \dots, l_n$ under which there is some other execution with l and l' adjacent: $p, m \xrightarrow{\hat{l_1}} \dots \xrightarrow{\hat{l_n}} \xrightarrow{\bar{l}} \xrightarrow{\bar{l'}}$ (where $\bar{l}$ and $\bar{l'}$ are like l and l' but perhaps with different vector clocks). Give such a condition, as liberal as you can, and explain why it has that property.

Unfortunately, I did not understand what the question was asking for.

2 Notes Page 103

37. Can you show all the conditions for *O2PL* are necessary, by giving for each an example that satisfies all the others and either is not serialisable or deadlocks?

- If a_i is $(\ell_j := n)$ or $!\ell_j = n$ then for some $k < i$ we have $a_k = \mathbf{lock}\ m_j$ without an intervening $\mathbf{unlock}\ m_j$.

$$\langle (\ell := 1; \ell := !\ell + 1) | (\ell := 2), \{s \mapsto 0\}, \emptyset \rangle$$

- For each j, the subsequence of $a_1, a_2, \dots$ with labels $\mathbf{lock}\ m_j$ and $\mathbf{unlock}\ m_j$ is a prefix of $((\mathbf{lock}\ m_j)(\mathbf{unlock}\ m_j))^*$. Moreover, if $\neg(e_k \xrightarrow{a})$ then the subsequence does not end in a $\mathbf{lock}\ m_j$.

$$
\begin{array}{l}
\langle \\
\textbf{lock}\ M_\ell; \ell := 0; \ell := !\ell + 1; \textbf{unlock}\ M_\ell \\
| \\
\textbf{unlock}\ M_\ell \\
| \\
\textbf{lock}\ M_\ell; \ell := 3; \textbf{unlock}\ M_\ell, \\
\{\ell \mapsto 0\}, \\
\{M_\ell \mapsto \textbf{false}\} \\
\rangle
\end{array}
$$

- If $a_i = \textbf{lock}\ m_j$ and $a_{i'} = \textbf{unlock}\ m_{j'}$ then $i < i'$

$$
\begin{array}{l}
\langle \\
\textbf{lock}\ M_\ell; \ell := 1; \textbf{unlock}\ M_\ell; \textbf{lock}\ M_\ell; \ell := !\ell + 1; \textbf{unlock}\ M_\ell; \\
| \\
\textbf{lock}\ M_\ell; \ell := 0; \textbf{unlock}\ M_\ell, \\
\{\ell \mapsto 0\}, \\
\{M_\ell \mapsto \textbf{true}\} \\
\rangle
\end{array}
$$

- If $a_i = \textbf{lock}\ m_j$ and $a_{i'} = \textbf{lock}\ m_{j'}$ and $i < i'$ then $j < j'$.

$$
\begin{array}{l}
\langle \\
\textbf{lock}\ M_1; \textbf{lock}\ M_2; \textbf{unlock}\ M_2; \textbf{uklock}\ M_1 \\
| \\
\textbf{lock}\ M_2; \textbf{lock}\ M_1; \textbf{unlock}\ M_2; \textbf{uklock}\ M_1 \\
\emptyset \\
\{M_1 \mapsto \textbf{true}, M_2 \mapsto \textbf{2}\} \\
\rangle
\end{array}
$$

39. Write a semantics for an extension of L1 with threads that are more like Unix threads (e.g. with thread ids, fork, etc…). Include some of the various ways Unix threads can exchange information.

$$
\begin{array}{l}
Process, p ::= (p|p)|(e, s) \\
Expression, e ::= \ldots \mid fork() \mid exit(e) \mid join()
\end{array}
$$

$$\frac{}{\langle fork(), s\rangle \to \langle((0, s)|(1, s))\rangle}\ \text{FORK}$$

$$\frac{\langle p_1\rangle \to \langle p_1'\rangle}{\langle p_1|p_2\rangle \to \langle p_1'|p_2\rangle}\ \text{PARL}$$

$$\frac{\langle p_1\rangle \to \langle p_1'\rangle}{\langle p_1|p_2\rangle \to \langle p_1'|p_2\rangle}\ \text{PARR}$$

$$\frac{\langle e, s\rangle \to \langle e', s'\rangle}{\langle exit(e), s\rangle \to \langle exit(e'), s'\rangle}\ \text{EXIT}$$

$$\frac{}{\langle((join(), s)|(exit(n), s'))\rangle \to \langle n, s\rangle}\ \text{JOIN}$$

$$\frac{}{\Gamma \vdash fork() : \text{int}}\ \text{FORK\_T}$$

$$\frac{}{\Gamma \vdash exit(e) : unit}\ \text{EXIT\_T}$$

$$\frac{}{\Gamma \vdash join() : \text{int}}\ \text{JOIN\_T}$$

3 Notes Page 109

40. Prove some of the other cases of the Congruence theorem for equivalence in L1.

Assume that $e \simeq^T_\Gamma e'$

case $C = \_\ op\ e_2$

Therefore $C[e]$ is of the form $e_1\ op\ e_2$

Recall the reduction rule $op1$

$$\frac{\langle e, s\rangle \to \langle e', s'\rangle'}{\langle e\ op\ e_2, s\rangle \to \langle e'\ op\ e_2, s'\rangle}$$

case $e \to^\omega \wedge e' \to^\omega$

The reduction rule $op1$ will be applied until e has been reduced to a value. However, by assumption $e \to^\omega$ – so will never be reduced to a value. This implies that the $op1$ reduction rule can be infinitely applied. Therefore $C[e] \to^\omega$.

Similarly, $C[e'] \to^\omega$.

Therefore $C[e] \simeq^T_\Gamma C[e']$ and the congruence property holds in this case.

case $e \to^* v \wedge e' \to^* v$

The reduction rule $op1$ will be applied until e has been reduced to a value. By assumption, this happens in a finite number of steps. Therefore $C[e] \to^* v\ op\ e_2$. Similarly, $C[e'] \to^* v\ op\ e_2$.

Since both expressions have reduced to the same expression, all further reductions will be the same. We can therefore conclude that $C[e] \simeq^T_\Gamma C[e']$ and the congruence property holds in this case.

Since the congruence property for $\simeq^T_\Gamma$ holds for both sub-cases, we can conclude that it holds in this case.

case **if** e_1 **then** $\_$ **else** e_3

Recall the reduction rules for if. Invariant of the content of e or e', both $C[e]$ and $C[e']$ will both evaluate the expression e_1 until it is reduced to a value.

case $e_1 \to^\omega$

In this case, the reduction rule $if1$ will be infinitely applied for both $C[e]$ and $C[e']$. Therefore $C[e] \to^\omega$ and $C[e'] \to^\omega$. Since both $C[e]$ and $C[e']$ loop infinitely: $C[e] \simeq^T_\Gamma C[e']$.

case $e_1 \to^*$ **true**

$$e_1 \to^* \textbf{true} \implies$$
$$C[e] \to^* \textbf{if true then } e \textbf{ else } e_2$$

In this case, the reduction rule $if2$ can be applied. Therefore $C[e] \to^* e$. Similarly, $C[e'] \to^* e'$. Since $e \simeq_\Gamma^T e'$, we can conclude that $C[e] \simeq_\Gamma^T C[e']$ – therefore contextual equivalence holds in this case.

case $e_1 \to^*$ **false**

$$e_1 \to^* \textbf{false} \implies$$
$$C[e] \to^* \textbf{if false then } e \textbf{ else } e_2$$

In this case, the reduction rule $if3$ can be applied – so $C[e] \to^* e_2$. Similarly, $C[e'] \to^* e_2$. Since both expressions have reduced to the same expression, they are equivalent – therefore $C[e] \simeq_\Gamma^T C[e']$ in this case.

Since contextual equivalence is proved for all subcases, it must hold in this case.

4 2017 Paper 6 Question 10

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2017p6q10.pdf

Let x range over a set X of identifiers, n range over the natural numbers $\mathbb{N}$ and s range over stores: total functions from X to $\mathbb{N}$.

Consider a language with the following abstract syntax.

$$e ::= n \mid x := e \mid !x \mid e_1; e_2$$

(a) Define a conventional deterministic small-step operational semantics $\langle e, s\rangle \to \langle e', s'\rangle$ for the language. Comment briefly on the choices you make.

The language does not contain **skip**. It is therefore necessary for sequences where the first expression is an integer to transition to the second sequence. It also makes the most logical sense for assignment to return the value. This permits ie chained assignment. The language uses left-to-right evaluation.

$$\frac{}{\langle x := n, s\rangle \to \langle n, s + \{x \mapsto n\}\rangle}\ \text{(assign1)}$$

$$\frac{\langle e, s\rangle \to \langle e', s'\rangle}{\langle x := e, s\rangle \to \langle x := e', s'\rangle}\ \text{(assign2)}$$

$$\frac{s(x) = n}{\langle !x, s\rangle \to \langle n, s\rangle}\ \text{(deref)}$$

$$\frac{}{\langle n; e, s\rangle \to \langle e, s\rangle}\ \text{(seq1)}$$

$$\frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle e_1; e_2, s\rangle \to \langle e_1'; e_2, s'\rangle}\ \text{(seq2)}$$

(b) If your language is deterministic and terminating, the operating semantics implicitly defines a more abstract semantics: we can regard each expression as a function over stores $[\![e]\!]$ that takes store s to the unique number n and store s' such that

$$\langle e, s\rangle \longrightarrow^* \langle n, s'\rangle \wedge \nexists e'', s''.\langle n, s'\rangle \longrightarrow \langle e'', s''\rangle$$

This language is quite limited in expressiveness. Describe, as clearly and precisely as you can, the set of functions from stores to (number, store) pairs that are expressible as $[\![e]\!]$ for some e.

The language has no ways of *modifying* numbers or branching. Informally, the value of every location in the final store after an expression has evaluated is either a value in the initial store or is hardcoded into the program.

Formally:

$$\forall e, s, n, s'.\langle e, s\rangle \to^* \langle n, s'\rangle \implies$$
$$(\forall y \in \mathrm{dom}(s').\exists x \in \mathrm{dom}(s).s(x) = s'(y)) \vee (\forall s''.\langle e, s''\rangle \to^* \langle n', s'''\rangle \implies s'(y) = s'''(y))$$
$$\wedge$$
$$(\exists x \in s.s(x) = n) \vee (\forall s''.\langle e, s''\rangle \to^* \langle n', s'''\rangle \implies n = n')$$

(c) the primitive contexts C for this language are expressions with a single hole:

$$C ::= x := \_ \mid e_1; \_ \mid \_; e_2$$

Write $C[e]$ for the expression resulting from replacing the hole in C by e.

Say a binary relation $\sim$ over expressions is a congruence if $e \sim e'$ implies $\forall C.C[e] \sim C[e']$.

Say a binary relation $\sim$ over expressions respects final values if $e \sim e'$ implies $\forall s_0, n, n', s, s'.(\langle e, s_0\rangle \longrightarrow \langle n, s\rangle \wedge \langle e', s_0\rangle \longrightarrow \langle n', s'\rangle) \implies n = n'$

Use your characterisation of part (b) to define an equivalence relation over expressions that is a congruence and respects final values. Explain briefly why it has those properties

$$e \sim e' \iff$$
$$\forall s_0.(\langle e, s_0\rangle \longrightarrow^* \langle n, s\rangle \wedge \langle e', s_0\rangle \longrightarrow^* \langle n', s'\rangle \implies n = n' \wedge s = s')$$

This equivalence respects final values. This language does not contain any loops and so every valid expression is guaranteed to terminate. When the expressions have reduced to an integer, this equivalence requires that integer to be the same. This is the definition of respecting final values. The equivalence also requires that any store the expressions evaluate to are the same. So the side effects of any two expressions which are equivalent are the same. So in any context: the return value and the effects are the same. So they are both contextually equivalent and respect final values.

(d) Define a terminating algorithm that, for any expressions e and e', computes whether $e \sim e'$ or not. Explain informally why it is correct.

Take two expressions. Consider the set of identifiers which occur in either expression. Place a total order on these – let x_n be the n^{th} identifier under this total order. Define two stores $s_e = \{(x_i, 2 \cdot i)\}$ and $s_o = \{(x_i, 2 \cdot i + 1)\}$.

Evaluate each expression with each store:

$$\langle e, s_1\rangle \longrightarrow^* \langle n_1, s_1'\rangle$$
$$\langle e', s_2\rangle \longrightarrow^* \langle n_2, s_2'\rangle$$
$$\langle e, s_1\rangle \longrightarrow^* \langle n_1', s_1''\rangle$$
$$\langle e', s_2\rangle \longrightarrow^* \langle n_2', s_2''\rangle$$

$$n_1 = n_1' \wedge n_2 = n_2' \wedge s_1 = s_1' \wedge s_2 = s_2'$$
$$\iff$$
$$e \sim e'$$

This algorithm evaluates expressions with an initial store with unique values. If the results of the expressions are the same, it repeats with another unique store to verify

there is no situation where e assigns a constant value which happens to be the same as the value in the location e' assigned or vice-versa.

If this algorithm finds $e \nsim e'$ then the two expressions produced different results in one of the tests. So there exists a counterexample to $e \sim e'$. Therefore $e \nsim e'$.

The language has no integer manipulation. All the machine can do is shuffle the contents of the state or assign them to constants. This is also true for the return value. Since $\forall x.s(x)$ is unique and the value of a location $s(x)$ after e executes is n then either e assigns $s(x)$ to the constant n or there exists some y such that e assigns $s(x) = s(y)$. By running the algorithm twice with disjoint stores, we can tell for each location ℓ whether the effect of an expression e is to assign it to a constant or the value of another location. We can also tell which constant or location this was. If $e \nsim e'$, then either their effects on the return value will be different or their effects on some store will be different. This test passes if and only if their effects on both the store and the return value are the same in all situations. Therefore, if $e \nsim e'$ then the test does not pass.

Combining this and the previous result:

$$e \sim e' \iff \text{the test passes}$$

5 2004 Paper 6 Question 11

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2004p6q11.pdf

L1 has the expression syntax

$$\begin{aligned} e &::= n \mid b \mid e_1 \; op \; e_2 \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \\ &\mid \ell := e \mid !\ell \mid \textbf{skip} \mid e_1; e_2 \mid \textbf{while } e_1 \textbf{ do } e_2 \end{aligned}$$

(a) Give the reduction rules for conditionals and while-loops.

$$\frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, s\rangle \to \langle \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3, s'\rangle} \text{ (if1)}$$

$$\frac{}{\langle \textbf{if true then } e_2 \textbf{ else } e_3, s\rangle \to \langle e_2, s\rangle} \text{ (if2)}$$

$$\frac{}{\langle \textbf{if false then } e_2 \textbf{ else } e_3, s\rangle \to \langle e_3, s\rangle} \text{ (if3)}$$

$$\frac{}{\langle \textbf{while } e \textbf{ do } e', s\rangle \to \langle \textbf{if } e \textbf{ then } e'; \textbf{while } e \textbf{ do } e' \textbf{ else skip}, s\rangle} \text{ (while)}$$

(b) Define semantic equivalence $e_1 \simeq_\Gamma^T e_2$ for L1.

Informally, either both e_1 and e_2 loop infinitely or they both terminate and their final values and stores are the same.

$$\begin{aligned} e_1 \simeq_\Gamma^T e_2 \iff \\ (\Gamma \vdash e_1 : T \wedge \Gamma \vdash e_2 : T \wedge \\ \forall s.\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(s) \implies \\ \langle e_1, s\rangle \longrightarrow^\omega \\ \wedge \langle e_2, s\rangle \longrightarrow^\omega \\ \vee \langle e_1, s\rangle \longrightarrow^* \langle v, s'\rangle \\ \wedge \langle e_2, s\rangle \longrightarrow^* \langle v, s'\rangle \end{aligned}$$

(c) For each of the following pairs, state whether they are semantically equivalent; if not, state a nontrivial condition on the subexpressions e, e_1, e_2, e_3 that makes them so, and explain informally why it suffices.

(i) $l \coloneqq 3; e \overset{?}{\simeq} e; l \coloneqq 3$

These are not equivalent, consider the case $e = l \coloneqq 0$.

A necessary and sufficient criteria is: for all stores s such that $l \in \mathrm{dom}(s)$. e assigns a value to l before any reads and the final access to l is a write setting l to 3. Note that this holds for no accesses to the variable.

(ii) $e; (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) \overset{?}{\simeq} \textbf{if } e_1 \textbf{ then } e; e_2 \textbf{ else } e; e_3$

These two expressions are not equivalent. This is because e is evaluated before e_1 in the first expression and e_1 is evaluated before e in the second expression.

$$e; \textbf{if } e_1 \textbf{ then skip else skip} \simeq^{T}_{\Gamma} \textbf{if } e_1 \textbf{ then skip else skip}; e$$

(iii) $e; (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) \overset{?}{\simeq} \textbf{if } e; e_1 \textbf{ then } e_2 \textbf{ else } e_3$

These are semantically equivalent. Since in all cases, the order in which the expressions is evaluated is the same.

(iv) $\textbf{while } !l \geq 0 \textbf{ do } (e_2; e_3) \overset{?}{\simeq} \textbf{if } !l \geq 0 \textbf{ then } e_2; (\textbf{while } !l \geq 0 \textbf{ do } (e_3; e_2)); e_3 \textbf{ else skip}$

These are both semantically equivalent. The predicate $l \geq 0$ is always true since $!l \in \mathbb{N}$ and all natural numbers are ≥ 0. So both expressions a never-ending sequence of reductions. So they are semantically equivalent.

Note that if the predicate $l \geq 0$ was not always true then the expressions would not be semantically equivalent.

6 2021 Paper 4 Question 9

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2021p4q9.pdf

(a) Suppose that $l : \mathrm{intref} \in \Gamma$. Now, consider the following program equivalence for L1:

$$(\text{if } !l \leq 0 \text{ then } e_1 \text{ else } e_2); e_3 \simeq^{\Gamma}_{\mathrm{unit}} (\text{it } !l \leq 0 \text{ then } e_1; e_3 \text{ else } e_2; e_3)$$

(i) Explain informally but carefully why this equivalence holds.

The order in which the subexpressions are evaluated is the same in both programs.

(ii) Using the definition of semantic equivalence, prove that this equivalence holds.

Split into cases on the truth of $!l \leq 0$:

case $!l \leq 0$

For the LHS program:

$$\begin{aligned} \langle (\text{if } !l \leq 0 \text{ then } e_1 \text{ else } e_2); e_3, s\rangle &\longrightarrow^2 \\ \langle (\text{if } \textbf{true} \text{ then } e_1 \text{ else } e_2); e_3, s\rangle &\longrightarrow \\ \langle e_1; e_3, s\rangle &\longrightarrow \end{aligned}$$

For the RHS program:

$$\begin{aligned} \langle (\text{if } !l \leq 0 \text{ then } ((e_1; e_3)) \text{ else } (e_2; e_3)), s\rangle &\longrightarrow^2 \\ \langle (\text{if } \textbf{true} \text{ then } ((e_1; e_3)) \text{ else } (e_2; e_3)), s\rangle &\longrightarrow \\ \langle e_1; e_3, s\rangle &\longrightarrow \end{aligned}$$

Since both programs reduce to the same program (and a program is trivially semantically equivalent to itself), we can conclude that the two original programs are semantically equivalent.

case $!l > 0$

For the LHS program:

$$\begin{aligned}\langle(\text{if } !l \leq 0 \text{ then } e_1 \text{ else } e_2); e_3, s\rangle &\longrightarrow^2 \\ \langle(\text{if } \mathbf{false} \text{ then } e_1 \text{ else } e_2); e_3, s\rangle &\longrightarrow \\ \langle e_2; e_3, s\rangle &\longrightarrow\end{aligned}$$

For the RHS program:

$$\begin{aligned}\langle(\text{if } !l \leq 0 \text{ then } ((e_1; e_3)) \text{ else } (e_2; e_3)), s\rangle &\longrightarrow^2 \\ \langle(\text{if } \mathbf{false} \text{ then } ((e_1; e_3)) \text{ else } (e_2; e_3)), s\rangle &\longrightarrow \\ \langle e_2; e_3, s\rangle &\longrightarrow\end{aligned}$$

In this case, both programs reduce to the same program. A program is trivially semantically equivalent to itself. Therefore the two programs are semantically equivalent.

(b) Now, consider the following *non*-equivalence

$$e_3; (\text{if } !l \leq 0 \text{ then } e_1 \text{ else } e_2) \simeq^{\Gamma}_{\text{unit}} (\text{if } !l \leq 0 \text{ then } e_3; e_1 \text{ else } e_3; e_2)$$

(i) Give a well-typed example exhibiting a counterexample of this equivalence.

$$e_1 = l \coloneqq 1 \qquad e_2 = l \coloneqq 2 \qquad e_3 = l \coloneqq 3 \qquad s = \{l \mapsto 0\}$$

In the LHS program, e_3 is executed. When the predicate ie evaluated, it is false. So the else-branch is taken and l is set to 2.

In the RHS program, the predicate is first evaluated and is decided to be true. So the if-branch is taken, which sets l to 1.

(ii) Give a statically decidable condition under which the transformation is valid.

The transformation is valid if e does not write to l. Let the predicate $\Phi(e, l)$ mean "e cannot write to l".

Define it inductively:

$$\begin{aligned}\Phi(v, l) &= \mathbf{true} \\ \Phi(l \coloneqq e, l) &= \mathbf{false} \\ \Phi(l' \coloneqq e, l) &= \Phi(e, l) \\ \Phi(\text{fn } x \Rightarrow e, l) &= \Phi(e, l) \\ \Phi(e\ e', l) &= \Phi(e, l) \wedge \Phi(e', l) \\ \Phi(\text{while } e \text{ do } e', l) &= \Phi(e, l) \wedge \Phi(e', l) \\ \Phi(\text{if } e_1 \text{ then } e_2 \text{ else } e_3, l) &= \Phi(e_1, l) \wedge \Phi(e_2, l) \wedge \Phi(e_3, l) \\ \Phi(e_1 \text{ op } e_2, l) &= \Phi(e_1, l) \wedge \Phi(e_2, l)\end{aligned}$$

The transformation is valid if $\Phi(e, l)$.