1 Notes Page 32

1. Write a program to compute the factorial of the integer initially in location ℓ_1.

```
⟨$\ell_2 := !\ell_1$;
 $\ell_3 := 1$;
 While($!\ell_2 \geq 1$)
 do(
     $\ell_4 := !\ell_2$;
     $\ell_5 := 0$;
     While($!\ell_4 \geq 1$)
     do(
         $\ell_5 := !\ell_5 + !\ell_3$;
         $\ell_4 := !\ell_4 + -1$
     )
     $\ell_3 := !\ell_5$;
     $\ell_2 := !\ell_2 + -1$
 ),
 {
     $\ell_1 \mapsto n$,
     $\ell_2 \mapsto 0$,
     $\ell_3 \mapsto 0$,
     $\ell_4 \mapsto 0$,
     $\ell_5 \mapsto 0$,
 }⟩
```

✓

3. Give full derivations of the first four reduction steps of the $\langle e, s\rangle$ of the first L1 example on slide 22

Label your rules

$$\dfrac{\dfrac{\text{l\_0 in dom(s)}}{\langle \ell_0 := 7, \{\ell_0 \mapsto 0, \ell_1 \mapsto 0\}\rangle \to \langle \text{skip}, \{\ell_0 \mapsto 7, \ell_1 \mapsto 0\}\rangle}}{\langle (\ell_0 := 7); (\ell_1 := (!\ell_0 + 2)), \{\ell_0 \mapsto 0, \ell_1 \mapsto 0\}\rangle \to \langle \text{skip}; (\ell_1 := (!\ell_0 + 2)), \{\ell_0 \mapsto 7, \ell_1 \mapsto 0\}\rangle}$$

$$\dfrac{\dfrac{}{\langle \text{skip}; e, \emptyset\rangle \to \langle e, \emptyset\rangle}}{\langle \text{skip}; (\ell_1 := (!\ell_0 + 2)), \{\ell_0 \mapsto 7, \ell_1 \mapsto 0\}\rangle \to \langle \ell_1 := (!\ell_0 + 2), \{\ell_0 \mapsto 7, \ell_1 \mapsto 0\}\rangle}$$

The store is not empty (the rule applies for any store)

$$\dfrac{\dfrac{\text{l\_0 in dom(s)} \qquad \text{s(l\_0) = 7}}{\langle !\ell_0, \{\ell_0 \mapsto 7\}\rangle \to \langle 7, \{\ell_0 \mapsto 7\}\rangle}}{\langle (\ell_1 := (!\ell_0 + 2)), \{\ell_0 \mapsto 7, \ell_1 \mapsto 0\}\rangle \to \langle (\ell_1 := (7 + 2)), \{\ell_0 \mapsto 7, \ell_1 \mapsto 0\}\rangle}$$

You're missing a step:
using rule assign2 to
evaluate the RHS of :=

$$\dfrac{\dfrac{}{\langle 7 + 2, \emptyset\rangle \to \langle 9, \emptyset\rangle}}{\langle (\ell_1 := (7 + 2)), \{\ell_0 \mapsto 7, \ell_1 \mapsto 0\}\rangle \to \langle (\ell_1 := 9), \{\ell_0 \mapsto 7, \ell_1 \mapsto 0\}\rangle}$$

which updates the store and reduces to skip (then we're done)

✓

4. Adapt the implementation code to correspond to the two rules (op1b) and (op2b) on slide 44. Give some test cases that distinguish between the original and the new semantics.

To change the implementation such that it corresponds to op1b and op2b, we need to change only the match case for `Op` – this involves changing 9 characters:

```
...
let rec reduce (e, s) =
  match e with
   ...
   | Op (e1,opr,e2) ->
       (match (e1,opr,e2) with
       ...
       | (e1,opr,e2) -> (
           if (is_value e2) then
             (match reduce (e1,s) with
             | Some (e1',s') -> Some (Op(e1',opr,e2),s')        (* (op2b) *)
             | None -> None )
           else
             (match reduce (e2,s) with
             | Some (e2',s') -> Some(Op(e1,opr,e2'),s')         (* (op1b) *)
             | None -> None ) ) )
...
```

The following code samples would differentiate between the cases:

$$\langle !\ell + (\ell := 1; 1), \{\ell \mapsto 0\}\rangle$$
$$\langle \text{while } (!x \geq (x := !x - 1; 0)) \text{ do } (y := !y + !x)\rangle$$

✓

8. Give a type derivation for e on slide 32 with $\Gamma = \ell_1 : \text{intref}, \ell_2 : \text{intref}, \ell_3 : \text{intref}$.

   Done on next page.

$$\dfrac{\dfrac{\begin{array}{c}\Gamma \vdash 0 : \text{int} \\ \Gamma(\ell_2) = \text{intref}\end{array}}{\Gamma \vdash \ell_2 \coloneqq 0 : \text{int}}\,(\text{assign}) \qquad \dfrac{\dfrac{\Gamma \vdash -1 : \text{int} \quad \dfrac{\Gamma(\ell_1) : \text{intref}}{\Gamma \vdash !\ell_1 : \text{int}}\,(\text{deref})}{\Gamma \vdash !\ell_1 \geq 1 : \text{bool}}\,(\text{op}\geq) \quad \dfrac{\dfrac{\dfrac{\dfrac{\Gamma(\ell_2) : \text{intref}}{\Gamma \vdash !\ell_2 : \text{int}}\,(\text{deref}) \quad \dfrac{\Gamma(\ell_1) : \text{intref}}{\Gamma \vdash !\ell_1 : \text{int}}\,(\text{deref})}{\Gamma \vdash !\ell_1 + !\ell_2 : \text{int}}\,(\text{op}+)}{\Gamma \vdash \ell_2 \coloneqq !\ell_1 + !\ell_2 : \text{int}}\,(\text{assign}) \quad \dfrac{\dfrac{\Gamma \vdash -1 : \text{int} \quad \dfrac{\Gamma(\ell_1) : \text{intref}}{\Gamma \vdash !\ell_1 : \text{int}}\,(\text{deref})}{\Gamma \vdash !\ell_1 + -1 : \text{int}}\,(\text{op}+)}{\Gamma \vdash \ell_1 \coloneqq !\ell_1 + -1 : \text{unit}}\,(\text{assign})}{\Gamma \vdash \ell_2 \coloneqq !\ell_2 + !\ell_1; \ell_1 \coloneqq !\ell_1 + -1 : \text{unit}}\,(\text{seq})}{\Gamma \vdash \text{While}(!\ell_1 \geq 1)\ \text{do}\ (\ell_2 \coloneqq !\ell_2 + !\ell_1; \ell_1 \coloneqq !\ell_1 + -1) : \text{unit}}\,(\text{while})}{\Gamma \vdash \ell_2 \coloneqq 0; \text{While}(!\ell_1 \geq 1)\ \text{do}\ (\ell_2 \coloneqq !\ell_2 + !\ell_1; \ell_1 \coloneqq !\ell_1 + -1) : \text{unit}}\,(\text{seq})$$

✓

General comment: try to stick to mathematical notation, words don't capture all the subtleties very well

9. Does Type preservation hold for the variant language with rules `assign1′` and `seq1′` on slide 45? If not, give an example and show how type rules could be adjusted to make it true.

   Type preservation does not hold for the variant language with rules `assign1′` and `seq1′`. This is because the type returned from assignment is still `unit`.

   In the modified language, programs such as $\langle \ell := (\ell := 0) + 1, \{\ell \mapsto 0\}\rangle$ are valid. However, the typing rules would reject this program! The type of $\ell := 0$ would be `unit` – we cannot add a value of type unit and an integer so the program would be rejected under the previous typing rules..

   This can be resolved by changing the type derivation rule for `assign1′` to:

$$\frac{\Gamma(\ell) = \mathrm{intref} \quad \Gamma \vdash e : \mathrm{int}}{\Gamma \vdash \ell := e : \mathrm{int}}$$

✓

2 Notes Page 49

12. Without looking at the proof in the notes, do the cases of the proof of Theorem 1 (Determinacy) for e_1 op e_2 and while e_1 do e_2.

    We can prove Determinacy using Structural induction. on the expression tree

    Define $\Phi(e)$ as the expression e is deterministic:

Very neat presentation!

$$\Phi(e) \triangleq \forall e, e', s, s', s''(\langle e, s\rangle \rightarrow \langle e', s'\rangle) \wedge (\langle e, s\rangle \rightarrow \langle e'', s''\rangle) \implies (e' = e'' \wedge s' = s'')$$

Case e_1 op e_2

We must prove that:

$$\forall e_1, e_2 . \Phi(e_1) \wedge \Phi(e_2) \implies \Phi(e_1 \text{ op } e_2)$$

This can be split into four cases:

Case $e_1 \notin \mathbb{V}$

Since one of the premises of op2 is $e_2 \in \mathbb{V}$, we cannot apply the rule op2. Therefore the only rule we might be able to apply is op1.

Case $\forall s . \langle e_1, s\rangle \not\rightarrow$

This violates one of the premises of op1 and therefore we cannot reduce e using op1. Therefore e cannot be reduced and the left hand side of the implication does not hold. Therefore $\Phi(e)$.

✓

Case $\exists e_1', s, s' . \langle e_1, s\rangle \rightarrow \langle e_1', s'\rangle$

e_1 can be reduced. However, since one of the assumptions of structural induction is the determinacy of subexpressions. Therefore $\Phi(e_1)$. Therefore any reduction on e_1 is deterministic. Since the only rule we can apply is op1, the reduction is therefore deterministic.

Could be more precise /rigorous but that's correct ✓

Case $e_1 \in \mathbb{V} \wedge e_2 \notin \mathbb{V}$

An analogous argument holds as in the case of $e_1 \notin \mathbb{V}$, except using conditioning on e_2 and using the reduction rule op2.

✓

Case $e_1 \in \mathbb{Z} \wedge e_2 \in \mathbb{Z}$

Case op is $+$

In this case, the premise of op+ is met. However, no other premise is met and therefore the only reduction we can perform is using the rule op+. Since op+ is deterministic, the reduction is deterministic in this case.

✓

Case op is $\geq$

In this case, the premise of op$\geq$ is met. However, no other premise is met and therefore the only reduction we can perform is using the rule op$\geq$. Since op$\geq$ is deterministic, the reduction is deterministic in this case.

✓

Since reduction for e_1 op e_2 is deterministic in all cases; we can conclude that reduction when e_1 op e_2 is deterministic.

Case while e_1 do e_2

The only rule which is applicable to while is the rule while. Since the rule while is deterministic, the reduction from while must be deterministic. Therefore $\Phi(\text{while } e_1 \text{ do } e_2)$ Can you prove it using the definition?

13.5 Flesh out the statements of Inversion for the operational semantics and type system. Prove them by rule induction.

this question is a bit weird, we'll go through it together

I wasn't too sure how to "prove" most of this. It was felt like any proof was identical to simple assertion that the premises of the reduction rule must have held before the rule was applied and therefore these facts must hold about e or e'.

If $\langle e, s\rangle \to \langle \hat{e}, \hat{s}\rangle$

For any reduction rule to be applied, prior to its application all of its premises must have held and $\langle e, s\rangle \to \langle \hat{e}, \hat{s}\rangle$ must be of the form of the conclusion of the reduction rule.)

3. (op2) Given the rule (op2):

$$\frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle n + e_1, s\rangle \to \langle n + e_1', s'\rangle}$$

For it to be applied, the premises must have been satisfied and $\langle e, s\rangle \to \langle \hat{e}, \hat{s}\rangle$ must be in the form of the conclusion. We can therefore infer that there exists n, e_1, e_1' such that $e = n$ op e_1, $\hat{e} = n$ op e_1' and $\langle e_1, s\rangle \to \langle e_1', \hat{s}\rangle$.

4. (op$\geq$) For (op$\geq$) to have been applied, we can the premises must have been satisfied before it was applied and therefore there exists n_1, n_2 and b such that $e = n_1 \geq n_2$, $\hat{e} = b$, $\hat{s} = s$ and $b = n_1 \geq n_2$.

5. (deref) For (deref) to have been applied, we can the premises must have been satisfied before it was applied and therefore there exists ℓ, n such that $\ell \in \text{dom}(s)$, $s(\ell) = n$, $e = !\ell$, $\hat{e} = n$ and $s = \hat{s}$.

6. (assign1) For (assign1) to have been applied, we can the premises must have been satisfied before it was applied and therefore there exists ℓ and n such that $e = \ell := n$, $\hat{e} = \textbf{skip}$ and $\hat{s} = s + \{\ell \mapsto n\}$.

7. (assign2) For (assign2) to have been applied, we can the premises must have been satisfied before it was applied and therefore there exists ℓ, e_1 and e_1' such that $e = \ell := e_1$, $\hat{e} = \ell := e_1'$ and $s = \hat{s}$.

8. (if1) For (if1) to have been applied, we can the premises must have been satisfied before it was applied and therefore there exists e_2, e_3 such that $e =$ **if** true then e_2 else e_3, $\hat{e} = e_2$ and $s = \hat{s}$.

9. (if2) For (if2) to have been applied, we can the premises must have been satisfied before it was applied and therefore there exists e_2, e_3 such that $e =$ **if** false then e_2 else e_3, $\hat{e} = e_3$ and $s = \hat{s}$.

10. (if3) For (if3) to have been applied, we can the premises must have been satisfied before it was applied and therefore there exists e_1, e_2, e_3, e_1' such that $e =$ if e_1 then e_2 else e_3, $\langle e_1, s\rangle \to \langle e_1', s'\rangle$, $\hat{e} =$ if e_1' then e_2 else e_3 and $s = \hat{s}$.

11. (while) For (while) to have been applied, we can the premises must have been satisfied before it was applied and therefore there exists e_1, e_2 such that $e = \text{while } e_1 \text{ do } e_2$, $\hat{e} = \text{if } e_1 \text{ then } (e_2; \text{while } e_1 \text{ do } e_2) \text{ else } \textbf{skip}$ and $s = \hat{s}$.

14. Complete the proof of Theorem 2 (Progress)

Similar to the proof stub in the notes, define $\Phi(\Gamma, e, T)$ as follows:

$$\Phi(\Gamma, e, T) \triangleq \forall s. \text{dom}(\Gamma) \subseteq \text{dom}(s) \implies \text{value}(e) \vee (\exists e', s'. \langle e, s \rangle \rightarrow \langle e', s' \rangle)$$

Case (assign) Recall the rule

$$\frac{\Gamma \vdash \ell : \text{intref} \qquad \Gamma \vdash e : \text{int}}{\ell := e : \text{unit}}$$

Assume $\Phi(\Gamma, \ell, \text{intref})$ and $\Phi(\Gamma, e, \text{int})$

We are now required to prove $\Phi(\Gamma, \ell := e, \text{int})$. ✓

case value(e) You can only do this case split because of the assumption

By assumption:

$$\begin{gathered}\Gamma \vdash \ell : \text{intref} \wedge \text{dom}(\Gamma) \subseteq \text{dom}(s) \implies \\ \ell \in \text{dom}(\Gamma) \wedge \text{dom}(\Gamma) \subseteq \text{dom}(s) \implies \\ \ell \in \text{dom}(s)\end{gathered}$$

By assumption, e is of type integer. Therefore:

$$\text{value}(e) \implies e \in \mathbb{Z} \implies \exists n \in \mathbb{Z}. e = n$$ ✓

Using these results, both the premises for the reduction rule (assign1) are met:

$$\langle \ell := n, s \rangle \rightarrow \langle \textbf{skip}, s + \{\ell \mapsto n\} \rangle \text{ if } \ell \in \text{dom}(s)$$

Therefore there exists at least one e', s' (namely $\textbf{skip}, s + \{\ell \mapsto n\}$) such that $\langle \ell := e, s \rangle \rightarrow \langle e', s' \rangle$ ✓

case $\langle e, s \rangle \rightarrow \langle e', s' \rangle$

These are the premise for the reduction rule (assign2)

$$\frac{\langle e, s \rangle}{\langle \ell := e, s \rangle \rightarrow \langle \ell := e', s' \rangle}$$

Therefore in this case there is at least one e', s' such that:

$$\langle \ell := e \rangle \rightarrow \langle e', s' \rangle$$ ✓

Since we have conditioned on both disjunctions in the assumption $\Phi(\Gamma, e, \text{int})$, we can conclude that the result $\Phi(\Gamma, \ell := e, \text{unit})$ holds under the assumptions and therefore type preservation is closed under the typing rule (assign).

Case (skip) Recall the typing rule (skip):

$$\Gamma \vdash \textbf{skip} : \text{unit}$$

Since this rule has no premise, we have to show that $\forall \Gamma, e, T$ of the conclusion $\Phi(\Gamma, e, T)$. Since the conclusion only accepts one value of e namely **skip** and one type, namely unit; we can conclude that $e = \textbf{skip}$ and $T = \text{unit}$.

We are trying to prove value$(e) \vee (...)$. Since value(**skip**) and $e = \textbf{skip}$, the first disjunct is true.

Usually people don't include enough detail, but this is a bit too verbose for a very simple case :) ✓

Case (seq) Recall the rule

$$\frac{\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 ; e_2 : T}$$

By assumption value$(e_1) \vee (\exists e', s'.\langle e_1, s\rangle \rightarrow \langle e', s'\rangle)$. We shall perform case analysis on this:

case value(e_1)

Since $\Gamma \vdash e_1$: unit and the only value with type unit is **skip**, we can conclude that $e_1 = \textbf{skip}$. Recall the rule (seq1):

$$\langle \textbf{skip}; e_2, s\rangle \rightarrow \langle e_2, s\rangle$$

Since $e_1 = \textbf{skip}$, the expression is of this form and therefore we can apply the reduction rule (seq1). Therefore in this case the RHS of the disjunct is proven:

$$\exists e', s'.\langle e_1; e_2, s\rangle \rightarrow \langle e', s'\rangle$$

✓

case $\exists e', s' \langle e_1, s\rangle \rightarrow \langle e', s'\rangle$

This meets the premises for the reduction rule (seq2):

$$\frac{\langle e_1, s\rangle \rightarrow \langle e', s'\rangle}{\langle e_1; e_2, s\rangle \rightarrow \langle e_1'; e_2, s'\rangle}$$

Therefore we can apply the reduction rule (seq2) and the RHS of the disjunct is proven. ✓

Case (while) Recall the rule:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash \textbf{while}\ e_2\ \textbf{do}\ e_2 : \text{unit}}$$

Consider the reduction rule (while):

$$\langle \textbf{while}\ e_1\ \textbf{do}\ e_2, s\rangle \rightarrow \langle \textbf{if}\ e_1\ \textbf{then}\ (e_2; \textbf{while}\ e_1\ \textbf{do}\ e_2)\ \textbf{else}\ \textbf{skip}, s\rangle$$

There are no premises or conditions. Therefore all expressions of the form **while** e_1 **do** e_2 can use this reduction rule. Therefore there is at least one reduction rule which can be used and the RHS of the disjunct is proved: ✓

15. Complete the proof of Theorem 3 (Type Preservation)

Case (op2) Recall

Assuming n op e2 has type int and it reduces,
RTP the RHS also has type int

$$\frac{\langle e_2, s\rangle \rightarrow \langle e_2', s'\rangle}{\langle n \text{ op } e_2, s\rangle \rightarrow \langle n \text{ op } e_2, s'\rangle}$$

Imprecise

By the assumption of rule induction, $\Phi(e_2, s, e_2', s')$. The only rules which could have been applied to derive the type of an op is the (op) typing rule. Since both have premises $\Gamma \vdash e_2$: int, we can conclude that the previous type of e_2 was ✓

int. Since by assumption, $\Phi(e_2, s, e_2', s')$ we can conclude that e_2' also has type int under Γ. Therefore we can apply the typing rule (op):

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_1 : \text{int}}{\Gamma \vdash e_1 \text{ op } e_2 : \text{int}}$$

Therefore we can draw the conclusion that the type of e_1 op e_2' must be int – which is the same as the type of e_1 op e_2. Therefore the type has been preserved and $\Phi(e_1 \text{ op } e_2, s, e_1 \text{ op } e_2', s')$ ✓

Case (deref) Recall

$$\langle !\ell, s\rangle \rightarrow \langle n, s\rangle \quad \text{if } \ell \in \text{dom}(s) \text{ and } s(\ell) = n$$

We can use the typing rule (deref) on the LHS of this rule:

$$\frac{\Gamma(\ell) = \text{intref}}{\Gamma \vdash !\ell : \text{int}}$$

Therefore the LHS of this rule is of type int.

The RHS of this rule is n. Therefore we can apply the (int) typing rule:

$$\Gamma \vdash n : \text{int} \quad \text{for } n \in \mathbb{Z}$$

Therefore the RHS of this rule is also of type int. Since both sides of this rule are of type int, we can conclude that this rule preserves typing. ✓

Case (assign1) Recall:

$$\langle \ell := n, s\rangle \rightarrow \langle \textbf{skip}, s + \{\ell \mapsto n\}\rangle \quad textif\ \ell \in \text{dom}(s)$$

We can apply the (assign) typing rule to the LHS of this rule:

$$\frac{\Gamma \vdash \ell : \text{intref} \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash \ell := e : \text{unit}}$$

Therefore the LHS of this rule must be of type unit. We can then apply the typing rule (skip) to the RHS of the rule to derive the type of **skip**.

$$\Gamma \vdash \textbf{skip} : \text{unit}$$

Therefore the type of **skip** is unit. Since both sides of the expression have type unit, we can therefore conclude that the rule must preserve typing. ✓

Case (assign2) Recall:

$$\frac{\langle e, s\rangle \rightarrow \langle e', s'\rangle}{\langle \ell := e, s\rangle \rightarrow \langle \ell := e', s\rangle}$$

We can apply the typing rule (assign) to the LHS of the rule:

$$\frac{\Gamma \vdash \ell : \text{intref} \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash \ell := e : \text{unit}}$$

Therefore the LHS of the rule must always be of type unit.

By assumption of rule induction, $\Phi(e, s, e', s')$. This implies that e' also has type int. Therefore we can apply the typing rule (assign) to the RHS of the rule as well. This derives that the type of the conclusion is also unit.

Therefore both the original expression and the reduction have the same type (unit) and so the reduction rule (assign2) preserves type. ✓

Case (seq1) Recall:

$$\langle \textbf{skip}; e, s\rangle \rightarrow \langle e, s\rangle$$

We can apply the typing rule (seq) to the LHS of this rule:

$$\frac{\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1; e_2 : T}$$

Since the type of **skip** is unit, we can apply this rule. Therefore the type of **skip**; e_2 is the same as the type of e_2.

After applying the reduction rule (seq1), the expression is e_2. Therefore the type of the whole expression is e_2. Therefore the type of the expression prior to applying the reduction rule is the same as the type of the expression after applying the reduction rule. Hence (seq1) preserves type. ✓

Case (seq2) Recall:

$$\frac{\langle e_1, s\rangle \rightarrow \langle e_1', s'\rangle}{\langle e_1; e_2, s\rangle \rightarrow \langle e_1'; e_2, s'\rangle}$$

Also too wordy

By assumption, the expression is properly typed. Since there is only one typing rule (seq) which can be applied to ";", we must have applied (seq) to get the original type of the expression. Therefore the type of e_1 must be of type unit. By assumption, $\Phi(e_1, s, e_1', s')$. Therefore the type of e_1' must be the same as the type of e_1 – namely unit. This allows us to apply the reduction rule (assign). Assign concludes that the type of $e_1; e_2$ is the same as the type of e_2. Since e_2 was not reduced by (seq2) we can conclude that both before and after the reduction rule (seq2) is applied, the type of $e_1; e_2$ must be the type of e_2. Therefore the rule (seq2) preserves typing. ✓

Case (if1) Recall:

$$\langle \text{if true then } e_2 \text{ else } e_3, s\rangle \rightarrow \langle e_2, s\rangle$$

Using the typing rule (if) we can conclude that the type of the LHS of the rule is the type of e_2

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T}$$

Since the RHS of the typing rule is e_2, we can conclude that the type of the RHS of the reduction must also be the type of e_2. Since both the LHS and the RHS of the expression have the same type, we can conclude that the reduction rule (if1) preserves type. ✓

Case (if2) Recall:

$$\langle \text{if false then } e_2 \text{ else } e_3, s\rangle \rightarrow \langle e_3\rangle$$

Similar to above, we can use the typing rule (if) to prove that the type of the LHS of the reduction is the same as the type of e_3 – which is also the RHS of the reduction and therefore both sides of the reduction have the same type, meaning the reduction rule (if2) preserves types. ✓

Case (if3) Recall:

$$\frac{\langle e_1, s\rangle \rightarrow \langle e_1', s'\rangle}{\langle \text{if } e_1 \text{ then } e_2 \text{ else } e_3, s\rangle \rightarrow \langle \text{if } e_1' \text{ then } e_2 \text{ else } e_3, s'\rangle}$$

By assumption, the expression is well typed. Therefore there must be some typing rule which can be applied to it. There is only one typing rule which can be applied to expressions of this form: (if). One of the premises of (if) is that the type of e_1 is bool. Since, by assumption $\Phi(e_1, s, e_1', s')$ we know that e_1' must also have type bool. Therefore we can apply the typing rule (if):

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T}$$

Therefore, since we have not changed e_2 or e_3, the type of the if statement must be unchanged. Therefore, we can conclude that the typing rule (if3) preserves types. ✓

Case (while) Recall:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash \text{while } e_1 \text{ do } e_2 : \text{unit}}$$

Therefore the type of the expression before the reduction is unit. The while transition is as follows:

$$\langle \textbf{while } e_1 \textbf{ do } e_2, s\rangle \rightarrow \langle \textbf{if } e_1 \textbf{ then } (e_2; \textbf{while } e_1 \textbf{ do } e_2) \textbf{ else skip}, s\rangle$$

Therefore the expression will reduce to an if expression. We can now apply the (if) typing rule to derive the type of the expression:

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : T}$$

Consider e_3 – this is **skip** which has type unit. Therefore after reduction, the expression also has type unit. Therefore the (while) reduction preserves type. ✓

3 2012 Paper 6 Question 10

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2012p6q10.pdf

This question is about a simple functional programming language with the following syntax.

$$\begin{array}{ll} \textit{Expressions}: & e ::= \ x \mid \text{skip} \mid \text{fn } x : T \rightarrow e \mid e\ e' \\ \textit{Types}: & T ::= \ \text{unit} \mid T \rightarrow T' \end{array}$$

(a) Give rules defining a typing relation ($\vdash$) for this language.

You need to look up x in the context (it could be a unit -> unit, etc)

True for an arbitrary context, not just empty

$$\emptyset \vdash x : \text{unit}$$
$$\emptyset \vdash \text{skip} : \text{unit}$$
$$\{e \vdash T'\} \vdash \text{fn } x : T \rightarrow e : T \rightarrow T'$$
$$\{e \vdash T \rightarrow T', e' \vdash T\} \vdash e\ e' : T'$$

e is not a variable. You need to add premises to your rules

(b) Give a brief illustration of the following concepts: *free variables* and *closed expression.*

A free variable is a variable which is not bound by a lambda. In the example below y is a free variable:

$$\text{fn } x \rightarrow x + y$$

A closed expression is an expression with no free variables. ✓

(c) Give rules defining a transition relation ($\to$) for this language. Use call-by-value evaluation order and take care to say what the values are.

The values in this language are skip of type unit. Note that since this language is fully functional, there is no store and therefore the relation does not include a store.

✓

$$\cancel{\frac{}{\langle x\rangle \to \langle \text{skip}\rangle}}$$

Variables don't step

$$\frac{\langle e'\rangle \to \langle e''\rangle}{\langle e\ e'\rangle \to \langle e\ e''\rangle}$$

Need a rule to reduce the first argument once the second is a value

$$\frac{}{\langle (\text{fn } x \to e)\ v\rangle \to \langle \{v/x\}e\rangle}$$

4 2015 Paper 6 Question 10

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2015p6q10.pdf

Consider the following syntax:

Booleans $b \in \mathbb{B} = \{\textbf{true}, \textbf{false}\}$
Integers $n \in \mathbb{Z} = \{\dots, -1, 0, 1, \dots\}$
Variables $x \in \mathbb{X} = \{x, y, \dots\}$
Expressions $e ::= b \mid n \mid x \mid \textbf{fn}\ x \to e \mid e_1\ e_2 \mid \textbf{print}\ e \mid \textbf{skip}$
Considered up to alpha equivalence with x binding in e in $\textbf{fn}\ x \to e$.

The set of free variables of an expression fv(e) are defined in the normal way as follows:

$$\begin{aligned}
\text{fv}(b) &= \{\}\\
\text{fv}(n) &= \{\}\\
\text{fv}(\textbf{fn}\ y \to e) &= \text{fv}(e) - \{y\}\\
\text{fv}(e_1\ e_2) &= \text{fv}(e_1) \cup \text{fv}(e_2)\\
\text{fv}(\textbf{print}\ e) &= \text{fv}(e)\\
\text{fv}(\textbf{skip}) &= \{\}
\end{aligned}$$

(a) Define capture-avoiding substitution $\{e/x\}e'$.

Capture-avoiding substitution $\{e/x\}e'$ is the process of renaming (alpha converting) all bound variables in e' which are free in e to variable names which do not affect the binding graph. This is done to avoid "capturing" any free variables (assigning them values). The binding graph of both e and e' should remain unaffected by the substitution.

✓

You also need to define how it's computed

(b) Define a small-step right-to-left call-by-value operational semantics for this syntax. Your semantics should be expressed as a relation

$$e \xrightarrow{L} e'$$

Where the label L is either n (for a **print** of that integer) or τ (for an internal transition).

$$\frac{\langle e_2\rangle \xrightarrow{L} \langle e_2'\rangle}{\langle e_1\ e_2\rangle \xrightarrow[L]{} \langle e_1\ e_2'\rangle}\text{CL1}$$

Missing a rule for app congruence

$$\frac{}{\langle (\textbf{fn } x \rightarrow e)\ (n)\rangle \xrightarrow[\text{tau}]{} \langle \{n/x\}e\rangle}\text{CL2}$$

$$\frac{\langle e\rangle \xrightarrow{L} \langle e'\rangle}{\langle \textbf{print } e\rangle \xrightarrow[L]{} \langle \textbf{print } e'\rangle}\text{PRINT}$$

Missing a rule for printing n

(c) Explain how call-by-name semantics would differ, giving any changes required to the rules and giving an example expression that has different output in the two semantics (you should give its transitions in each but need not give their derivations).

Call-by-name semantics evaluate the argument to a function only when it is used. Therefore if an argument is not used, it will not be evaluated. Since this langauge is not purely functional (namely we can print) call-by-name and call-by-value do not always have the same behaviour.

To implement call-by-value, we would remove the CL1 and CL2 rules and add the following reduction rule: ✓

$$\frac{}{\langle (\textbf{fn } x \rightarrow e_1)\ (e_2)\rangle \rightarrow \langle \{e_2/x\}e_1\rangle}\text{CALL}$$

In the following example, call-by-value and call-by-name will behave differently:

$$\langle (\textbf{fn } x \rightarrow 1)(\textbf{print } 0)\rangle$$

Under call-by-value 0 is printed:

$$\begin{array}{ll} \langle (\textbf{fn } x \rightarrow 1)(\textbf{print } 0)\rangle & \xrightarrow{\text{CL2, 0}} \\ \langle (\textbf{fn } x \rightarrow 1)(\textbf{skip})\rangle & \xrightarrow{\text{CL1}} \\ \langle 1\rangle & \not\rightarrow \end{array}$$

Under call-by-name 0 is not printed:

$$\begin{array}{ll} \langle (\textbf{fn } x \rightarrow 1)(\textbf{print } 0)\rangle & \xrightarrow{\text{CALL}} \\ \langle 1\rangle & \not\rightarrow \end{array}$$

✓

5 2019 Paper 4 Question 8

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2019p4q8.pdf

Consider the following C-like language, tinyC. It has locally scoped mutable variables and functions that take a single argument. Its operational semantics is defined as a transition system over configurations $\langle e, E, s\rangle$ where E is an environment $\{x_1 \mapsto n_1, \ldots, x_j \mapsto n_j\}$, mapping the variable names currently in scope to their addresses and s is a store $\{n_1 \mapsto v_1, \ldots, n_k \mapsto v_k\}$ mapping each currently allocated address to either an integer n or **undef**. In this question n ranges over $0 \ldots 2^{63} - 1$. Programs p consist of finite sets of definitions with distinct names.

$$\begin{aligned} \textit{expression}, e &::= n \mid x \mid x = e' \mid \{\textbf{int } x; e\} \mid e_1; e_2 \mid f(e) \mid \textbf{undef} \mid \textbf{kill } x \\ \textit{definition}, d &::= \textbf{int } f(\textbf{int } x)e \end{aligned}$$

$$\frac{E(x) = n \quad s(n) = n'}{\langle x, E, s\rangle \rightarrow \langle n', E, s\rangle}\text{DEREF} \qquad \frac{E(x) = n \quad n \in \mathbf{dom}(s)}{\langle \mathbf{kill}\ x, E, s\rangle \rightarrow \langle 0, E\backslash x, s\backslash n\rangle}\text{KILL}$$

$$\frac{\langle e, E, s\rangle \rightarrow \langle e', E', s'\rangle}{\langle x = e, E, s\rangle \rightarrow \langle x = e', E', s'\rangle}\text{AS1} \qquad \frac{E(x) = n \quad s(n) = v}{\langle x = n', E, s\rangle \rightarrow \langle n', E, s + [n \mapsto n']\rangle}\text{AS2}$$

$$\frac{x \notin \mathbf{dom}(E) \quad n \notin \mathbf{dom}(s) \quad \neg\exists n' < n. n' \notin \mathbf{dom}(s)}{\langle \{\mathbf{int}\ x; e\}, E, s\rangle \rightarrow \langle e; \mathbf{kill}\ x, E + [x \mapsto n], s + [n \mapsto \mathbf{undef}]\rangle}\text{LOCAL}$$

$$\frac{\langle e_1, E, s\rangle \rightarrow \langle e_1', E', s'\rangle}{\langle e_1; e_2, E, s\rangle \rightarrow \langle e_1'; e_2, E', s'\rangle}\text{SEQ1} \qquad \frac{}{\langle n; e, E, s\rangle \rightarrow \langle e, E, s\rangle}\text{SEQ2}$$

$$\frac{\langle e, E, s\rangle \rightarrow \langle e', E', s'\rangle}{\langle f(e), E, s\rangle \rightarrow \langle f(e'), E', s'\rangle}\text{CL1} \qquad \frac{\mathbf{int} f(\mathbf{int}\ x)\{e\} \in p}{\langle f(n), E, s\rangle \rightarrow \langle \{\mathbf{int}\ x; (x = n; e)\}, E, s\rangle}\text{CL2}$$

(a) For the configuration ⟨ g(3), {}, {}⟩ and program **int** g(**int** y){{**int** z;z=y}}, give the sequence of 11 configurations it transitions to. For each transition, include the list of rule names involved in its derivation, but not the derivation itself.

$$\begin{array}{ll}
\langle\ \text{g(3)}, \{\}, \{\}\rangle & \xrightarrow{\text{CL2}} \\
\langle\ \{\mathbf{int}\ \text{y};\ (\text{y=3};\ \{\mathbf{int}\ \text{z;z=y}\})\}, \{\}, \{\}\rangle & \xrightarrow{\text{LOCAL}} \\
\langle\ \text{y=3};\ \{\mathbf{int}\ \text{z;z=y}\};\ \mathbf{kill}\ \text{y}, \{\text{y} \mapsto 0\}, \{0 \mapsto \mathbf{undef}\}\rangle & \xrightarrow{\text{SEQ1, AS2}} \\
\langle\ 3;\ \{\mathbf{int}\ \text{z;z=y}\};\ \mathbf{kill}\ \text{y}, \{\text{y} \mapsto 0\}, \{0 \mapsto 3\}\rangle & \xrightarrow{\text{SEQ2}} \\
\langle\ \{\mathbf{int}\ \text{z;z=y}\};\ \mathbf{kill}\ \text{y}, \{\text{y} \mapsto 0\}, \{0 \mapsto 3\}\rangle & \xrightarrow{\text{SEQ1, LOCAL}} \\
\langle\ \text{z=y};\ \mathbf{kill}\ \text{z};\ \mathbf{kill}\ \text{y}, \{\text{y} \mapsto 0, \text{z} \mapsto 1\}, \{0 \mapsto 3, 1 \mapsto \mathbf{undef}\}\rangle & \xrightarrow{\text{SEQ1, AS1, DEREF}} \\
\langle\ \text{z=3};\ \mathbf{kill}\ \text{z};\ \mathbf{kill}\ \text{y}, \{\text{y} \mapsto 0, \text{z} \mapsto 1\}, \{0 \mapsto 3, 1 \mapsto \mathbf{undef}\}\rangle & \xrightarrow{\text{SEQ1, AS2}} \\
\langle\ 3;\ \mathbf{kill}\ \text{z};\ \mathbf{kill}\ \text{y}, \{\text{y} \mapsto 0, \text{z} \mapsto 1\}, \{0 \mapsto 3, 1 \mapsto 3\}\rangle & \xrightarrow{\text{SEQ2}} \\
\langle\ \mathbf{kill}\ \text{z};\ \mathbf{kill}\ \text{y}, \{\text{y} \mapsto 0, \text{z} \mapsto 1\}, \{0 \mapsto 3, 1 \mapsto 3\}\rangle & \xrightarrow{\text{SEQ1, KILL}} \\
\langle\ 0;\ \mathbf{kill}\ \text{y}, \{\text{y} \mapsto 0\}, \{0 \mapsto 3\}\rangle & \xrightarrow{\text{SEQ2}} \\
\langle\ \mathbf{kill}\ \text{y}, \{\text{y} \mapsto 0\}, \{0 \mapsto 3\}\rangle & \xrightarrow{\text{KILL}} \\
\langle\ 0, \{\}, \{\}\rangle & \not\rightarrow
\end{array}$$

✓

(b) For each of the following, briefly explain the key points of its tinyC semantics and what it illustrates, referring to the transitions and rules, and to the relationship between tinyC and the full C languages, as appropriate.

(i) ⟨{**int** y;g(y)},{},{}⟩ this crashes in tinyC (reading undef); undefined behaviour in C

This example highlights tinyC's lack of variable shadowing. The program will firstly instantiate y. Then the function will be called. However, it will be unable to create a new variable y since y ∈ **dom**(E). The program will then hit undefined behaviour and stop. This is different to C; which has variable shadowing.

$$\begin{array}{ll}
\langle\{\mathbf{int}\ \text{y;g(y)}\},\{\},\{\}\rangle & \xrightarrow{\text{LOCAL}} \\
\langle\text{g(y)};\ \mathbf{kill}\ \text{y},\{\text{y} \mapsto 0\},\ \{0 \mapsto \mathbf{undef}\}\rangle & \xrightarrow{\text{SEQ1, CL2, DEREF}} \\
\langle\text{g}(\mathbf{undef});\ \mathbf{kill}\ \text{y},\{\text{y} \mapsto 0\},\ \{0 \mapsto \mathbf{undef}\}\rangle & \xrightarrow{\text{SEQ1, CL1}} \\
\langle\{\mathbf{int}\ \text{y;y=}\mathbf{undef}\};\ \mathbf{kill}\ \text{y},\{\text{y} \mapsto 0\},\{0 \mapsto \mathbf{undef}\}\rangle & \not\rightarrow
\end{array}$$

(ii) ⟨{**int** y; 4};y,{},{}⟩

This example demonstrates local scoping in tinyC. Since the variable y was declared inside a scope, it cannot be accessed outside of this scope. The attempt to dereference y outside the scope in which y is valid will will result in undefined behaviour at runtime.

C also has local scoping. However the behaviour is slightly different; accessing the variable y outside the scope in which it is defined would fail at compile time rather than runtime.

✓

(iii) ⟨h(5),{},{}⟩, with the program **int** h (**int** y){y=6;y}

This example shows how tinyC does not have return variables. Although tinyC functions have types, they have no way to return any value. In order for any function in tinyC to do anything, it must mutate the state of a variable which was declared outside its scope (the tinyC equivalent of global variables) – which is a likely source of bugs.

This is different to C, where functions can have return values.

Can mutate function arguments
(but this remains local to function)

This could be resolved by changing the semantics of KILL to:

$$\frac{E(x) = n \quad n \in \mathbf{dom}(s) \quad s(n) = n'}{\langle \mathbf{kill}\ x, E, s\rangle \rightarrow \langle n', E\backslash x, s\backslash n\rangle}\text{KILL'}$$

This change would allow functions to return values – their return values would be the value stored in the argument which was provided by them. Since tinyC only has integer types, this would be acceptable and type safe.

$$\begin{array}{ll}
\langle \text{h(5)},\{\},\{\}\rangle & \xrightarrow{\text{CL2}} \\
\langle \{\mathbf{int}\ \text{y};\ (\text{y=6};\ \text{y})\},\{\},\{\}\rangle & \xrightarrow{\text{LOCAL}} \\
\langle \text{y=6};\ \text{y};\ \mathbf{kill}\ \text{y},\{\text{y}\mapsto 0\},\ \{0\mapsto \mathbf{undef}\}\rangle & \xrightarrow{\text{SEQ1, ASSIGN}} \\
\langle 6;\ \text{y};\ \mathbf{kill}\ \text{y},\{\text{y} \mapsto 0\},\ \{0 \mapsto \mathbf{undef}\}\rangle & \xrightarrow{\text{SEQ2}} \\
\langle \text{y};\ \mathbf{kill}\ \text{y},\{\text{y} \mapsto 0\},\ \{0 \mapsto 6\}\rangle & \xrightarrow{\text{SEQ1, DEREF}} \\
\langle 6;\ \mathbf{kill}\ \text{y},\{\text{y} \mapsto 0\},\ \{0 \mapsto 6\}\rangle & \xrightarrow{\text{SEQ2}} \\
\langle \mathbf{kill}\ \text{y},\{\text{y} \mapsto 0\},\ \{0 \mapsto 6\}\rangle & \xrightarrow{\text{KILL}} \\
\langle 0,\{\},\{\}\rangle & \not\rightarrow
\end{array}$$

(iv) ⟨{**int** y;(y=3;{**int** y;y=4});y},{},{}⟩

This code, again highlights the lack of variable shadowing in tinyC. This code will create a variable y, assign a value to 3 and then hit undefined behaviour – a premise for LOCAL is that x $\notin$ **dom**(E); however as y has been defined, this will not hold. Therefore the semantics do not allow tinyC to make any transitions.

This can be resolved by adding alpha renaming to the language.

This is in stark contrast to C, which fully implements variable shadowing – in C we can have any number of variables with the same name as long as they are declared in different scopes.

✓