Harry Langford
hjel2@cam.ac.uk

1 Warmup Questions

1. Write down the rules for introduction and elimination of abstract data types in System F (i.e. *pack* and let *pack*). Explicitly specify the "client code" that doesn't get to see the abstracted type.

$$\frac{\Theta \vdash A \text{ type} \qquad \Theta \vdash \Gamma \text{ ctx} \qquad \Theta; \Gamma, x : \alpha \vdash e : B}{\Theta; \Gamma \vdash \mathit{pack}_{\alpha.B}(A, e) : \exists\alpha.\, B}(\mathit{pack})$$

$$\frac{\Theta; \Gamma \vdash \mathit{pack}_{\alpha.B}(A, e) : \exists\alpha.\, B \qquad \Theta; \Gamma, x : B \vdash e' : C}{\Theta; \Gamma \vdash \text{let } \mathit{pack}(\alpha, x) = \mathit{pack}_{\alpha.B}(A, e) \text{ in } \bigcirc\!\!\!\!e' : C}(\mathit{letpack})$$ ✓

The 'client' code

2. What do σ and Σ represent?

   σ represents the store. While Σ represents the store type. ✓

3. Write down the typing rules and operational semantics for the imperative additions to STLC.

   Typing Rules:

   We add specific rules for interacting with the store. We then add typing rules for typing the store. We then add a rule for using both of these to type a context. Note that we must also modify all existing typing rules to use a store typing context as well.

   Interacting with the store:

$$\frac{\Gamma; \Sigma \vdash e : \text{ref } T}{\Gamma; \Sigma \vdash !e : T} \qquad \frac{\Gamma; \Sigma \vdash e : T}{\Gamma; \Sigma \vdash \text{new } e : \text{ref } T}$$

$$\frac{\Gamma; \Sigma \vdash e : T \qquad \Gamma; \Sigma \vdash \ell : \text{ref } T}{\Gamma; \Sigma \vdash \ell := e : \text{unit}} \qquad \frac{\ell : T \in \Sigma}{\Gamma; \Sigma \vdash \ell : \text{ref } T}$$

   Modifying existing typing rules:

$$\frac{\Gamma, x : A; \Sigma \vdash e : B}{\Gamma; \Sigma \vdash \lambda x : A.\, e : A \to B} \qquad \frac{\Gamma; \Sigma \vdash e : A \to B \qquad \Gamma; \Sigma \vdash e' : A}{\Gamma; \Sigma \vdash e\ e' : B}$$

$$\cdots$$

   Typing the Store:

$$\frac{}{\cdot \vdash \cdot : \cdot} \qquad \frac{\Sigma \vdash \sigma : \Sigma' \qquad \Sigma \vdash v : T}{\Sigma \vdash (\sigma, \ell : v) : (\Sigma', \ell : T)}$$

   Typing Configurations:

$$\frac{\Sigma \vdash \sigma : \Sigma \qquad \cdot; \Sigma \vdash e : T}{\langle \sigma, e \rangle : (\Sigma, T)}$$ ✓

   Reduction Rules:

   We start by making all reduction rules operate on configurations of the form $\langle \sigma, e \rangle$. This requires modifying all existing rules to make them carry a σ in all premises and conclusions. Next, we add specific rules for interacting with the store.

Interacting with the Store:

$$\frac{(\ell, v) \in \sigma}{\langle \sigma, !\ell\rangle \rightsquigarrow \langle \sigma, v\rangle} \qquad \frac{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma, e'\rangle}{\langle \sigma, !e\rangle \rightsquigarrow \langle \sigma, !e'\rangle}$$

$$\frac{}{\langle \sigma, \ell := v\rangle \rightsquigarrow \langle (\sigma, \ell : v), ()\rangle} \qquad \frac{\langle \sigma, e_1\rangle \rightsquigarrow \langle \sigma, e_1'\rangle}{\langle \sigma, e_1 := e_2\rangle \rightsquigarrow \langle \sigma, e_1' := e_2\rangle} \qquad \frac{\langle \sigma, e_2\rangle \rightsquigarrow \langle \sigma, e_2'\rangle}{\langle \sigma, \ell := e_2\rangle \rightsquigarrow \langle \sigma, \ell := e_2'\rangle}$$

$$\frac{\ell \notin \sigma}{\langle \sigma, \text{new } v\rangle \rightsquigarrow \langle (\sigma, \ell : v), ()\rangle} \qquad \frac{\langle \sigma, e\rangle \rightsquigarrow \langle \sigma, e'\rangle}{\langle \sigma, \text{new } e\rangle \rightsquigarrow \langle \sigma, \text{new } e'\rangle}$$

Threading σ through all other reduction rules:

$$\frac{\langle \sigma, e_1\rangle \rightsquigarrow \langle \sigma, e_1'\rangle}{\langle \sigma, e_1\ e_2\rangle \rightsquigarrow \langle \sigma, e_1'\ e_2\rangle} \qquad \frac{}{\langle \sigma, (\lambda x. e)\ v\rangle \rightsquigarrow \langle \sigma, [v/x]e\rangle} \ldots$$

✓

4. Explain the reason to separate typing logic for pure and impure terms when using monads to track effects and explain how they are linked together.

   We separate terms based on whether they have side-effects or not. Those terms which do have side effects (impure) have monadic types. Those terms which don't (pure); do not have monadic types. This allows the type system to determine whether evaluating a particular expression will have any side effects. This is useful when considering compiler optimisations as we can determine what can be done in parallel or lazily; and what has to be done in-order. Also: we can prove termination for the pure parts.

2 Regular Questions

1. For each of the following PLC (System F) typing judgements, is there a PLC type A_i that makes the judgement provable? In each case, give a type A_i and a typing derivation for, or explain why a typing derivation cannot exist.

   (a) $\cdot;\cdot \vdash \lambda x : (\forall \alpha. \alpha). (\Lambda \beta. x\ \beta) : A_1$

   $A_1 = (\forall \alpha. \alpha) \to \forall \beta. \beta$ makes the judgement provable.

$$\frac{\dfrac{\dfrac{\dfrac{}{\beta; x : \forall \alpha. \alpha. x : \forall \alpha. \alpha} \quad \dfrac{}{\beta; x : \forall \alpha. \alpha. \beta : \beta}}{\beta; x : \forall \alpha. \alpha. x\ \beta : \beta}}{\cdot; x : \forall \alpha. \alpha \vdash \Lambda \beta. x\ \beta : \forall \beta. \beta}}{\cdot;\cdot \vdash \lambda x : (\forall \alpha. \alpha). (\Lambda \beta. x\ \beta) : (\forall \alpha. \alpha) \to \forall \beta. \beta}$$

Figure 1: Typing derivation for 1a ✓

   (b) $\cdot;\cdot \vdash \Lambda \alpha. \lambda x : \alpha. \Lambda \beta. x\ \beta : A_2$

   There is no A_2 such that the typing derivation is provable. The reason for this is that we are applying β to a term of type α. But α does not have to take any type argument. So if this has any type then we allow the application of types to terms which do not take types (such as values). ✓

   (c) $\cdot;\cdot \vdash \lambda x : A_3. \Lambda \alpha. (x\ (\alpha \to \alpha)\ (x\ \alpha)) : A_3 \to \forall \beta. \beta$

   There is no A_3 such that the the typing derivation is provable.

   By repeated inversion on the LHS of the function application, we can see that $A_3 = \forall \alpha. X \to \alpha$ and $\forall \alpha. X$ for some X. However, notice that A_3 is bound before

α. So α is *not visible* to A_3 when it is bound – and it is only passed $\alpha \to \alpha$. Therefore such a type is impossible! The conclusion is right but I think the argument isn't, let's discuss in the SV.

(d) $\cdot; \cdot \vdash \lambda x : A_4. \Lambda\alpha. (x\ (\alpha \to \alpha)\ (x\ \alpha)) : A_4 \to \forall\alpha. (\alpha \to \alpha)$

$A_4 = \forall\beta. \beta \to \beta$ makes the judgement provable.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha; x : \forall\beta.\beta\to\beta \vdash x : \forall\beta.\beta\to\beta} \quad \dfrac{}{\alpha \vdash \alpha\to\alpha \text{ type}}}{\alpha; x : \forall\beta.\beta\to\beta \vdash x\ (\alpha\to\alpha) : (\alpha\to\alpha)\to(\alpha\to\alpha)} \quad \dfrac{\dfrac{}{\alpha; x : \forall\beta.\beta\to\beta \vdash x : \forall\beta.\beta\to\beta} \quad \dfrac{}{\alpha \vdash \alpha \text{ type}}}{\alpha; x : \forall\beta.\beta\to\beta \vdash (x\ \alpha) : \alpha\to\alpha}}{\dfrac{\alpha; x : \forall\beta.\beta\to\beta \vdash x\ (\alpha\to\alpha)\ (x\ \alpha) : \alpha\to\alpha}{\cdot; x : \forall\beta.\beta\to\beta \vdash \Lambda\alpha.(x\ (\alpha\to\alpha)\ (x\ \alpha)) : \forall\alpha.(\alpha\to\alpha)}}}{\cdot;\cdot \vdash \lambda x : \forall\beta.\beta\to\beta.\Lambda\alpha.(x\ (\alpha\to\alpha)\ (x\ \alpha)) : (\forall\beta.\beta\to\beta)\to\forall\alpha.(\alpha\to\alpha)}$$

Figure 2: Typing derivation for 1d ✓

(e) $\cdot; \cdot \vdash \Lambda\alpha. \lambda x : A_5. (x\ (\alpha \to \alpha)\ (x\ \alpha)) : \forall\alpha. (\alpha \to \alpha)$

There is no type A_5 which makes the judgement provable. Consider the type $\forall\alpha. \alpha \to \alpha$. Since the term first take $x : A_5$ as an argument, we are constrained that if there is a type A_5 then we must have $A_5 = \alpha$. But we then pass the types $(\alpha \to \alpha)$ to x. This places additional constraints on its type meaning that we have A_5 cannot possibly be the type α! So we have a contradiction and therefore there is no type A_5 which makes the judgment provable ✓

2. Complete Exercises 2, 3 and 4 from Lecture 5:

Exercises 2 and 3 confused me!

(2) Define a Church encoding for the unit type.

$$\mathsf{unit} = 1$$

(3) Define a church encoding for the empty type.

$$\mathsf{empty} = 0$$

(4) Define a Church encoding for binary trees, corresponding to the ML datatype

`type tree = Leaf | Node of tree * X * tree`

Let T be the type of a tree, N be the type of a node and L be the type of a leaf:

L is the unit type:

$$L = 1$$

N is a triple: Not quite. Will discuss in SV.

$$N = \forall\alpha. (X \to Y \to Z \to \alpha) \to \alpha$$

T is a sum of L or N:

$$T = L + N$$

Where $L + N$ can be defined using the definition of sum types given in the notes.

3. The *Church numerals* are defined as follows:

$$c_0 = \Lambda\alpha. \lambda z : \alpha. \lambda s : \alpha \to \alpha. z$$
$$c_1 = \Lambda\alpha. \lambda z : \alpha. \lambda s : \alpha \to \alpha. s\ z$$
$$c_2 = \Lambda\alpha. \lambda z : \alpha. \lambda s : \alpha \to \alpha. s(s\ z)$$
$$c_3 = \Lambda\alpha. \lambda z : \alpha. \lambda s : \alpha \to \alpha. s(s(s\ z))$$

The successor function that takes c_n to c_{n+1} is defined in the lectures as

$$suc(n) \overset{def}{=} \Lambda\alpha.\,\lambda z:\alpha.\,\lambda s:\alpha\to\alpha.\,s(n\ \alpha\ z\ s)$$

Using just the variables α, n, s and z, find another way to define the successor function, suc on Church numerals.

$$suc_2(n) \overset{def}{=} \Lambda\alpha.\,\lambda z:\alpha.\,\lambda s:\alpha\to\alpha.\,n\ \alpha\ (s\ z)\ s$$ ✓

4. Complete Exercises 1, 2 and 3 from Lecture 6.

   (1) Prove the other direction of the closure property for the $\Theta \vdash \forall\alpha.\,A$ type case.

   Closure (for $\forall\alpha.\,A$): if θ is an interpretation of Θ then $[\![\Theta \vdash \forall\alpha.\,A\ \text{type}]\!]\theta$ is a semantic type.

   To prove this, we have to prove that $[\![\Theta \vdash \forall\alpha.\,A\ \text{type}]\!]\theta$ is closed and all terms in it halt. The closure property of a Semantic Type X is $e \rightsquigarrow e' \implies e' \in X \iff e \in X$. We have to prove the direction $\Longleftarrow$.

$e \rightsquigarrow e'$	Assumption	(1)
$e \in [\![\Theta \vdash \forall\alpha.\,A\ \text{type}]\!]\theta$	Assumption	(2)
$\forall(C,X).\,e\ C \in [\![\Theta,\alpha \vdash A\ \text{type}]\!](\theta, X/\alpha)$	Definition of 2	(3)

Don't forget to show e' halts.

   Fix (C,X) for arbitrary (C,X)

$e\ C \in [\![\Theta,\alpha \vdash A\ \text{type}]\!](\theta, X/\alpha)$	by 3	(4)
$e\ C \rightsquigarrow e'\ C$	CongForall on 1	(5)
$e'\ C \rightsquigarrow e'\ C$	Induction on 4 and 5	(6)
$\forall(C,X).\,e'\ C \in [\![\Theta,\alpha \vdash A\ \text{type}]\!](\theta, X/\alpha)$	by 6 since (C,X) was arbitrary	(7)
$e' \in [\![\Theta \vdash A\ \text{type}]\!]\theta$	by 7	(8)

✓

   (2) Prove the other direction of the substitution property for the $\Theta \vdash \forall\alpha.\,A$ type case.

   Substitution (for $\forall\alpha.\,A$): $[\![\Theta,\alpha \vdash \forall\beta.\,B\ \text{type}]\!](\theta, [\![\Theta \vdash A\ \text{type}]\!]) = [\![\Theta \vdash [A/\alpha](\forall\beta.\,B)\ \text{type}]\!]\theta$.

$e \in [\![\Theta \vdash [A/\alpha](\forall\beta.\,B)\ \text{type}]\!]\theta$	Assumption	(9)
$e \in [\![\Theta \vdash \forall\beta.\,[A/\alpha]B\ \text{type}]\!]\theta$	Definition of substitution	(10)
$\forall(C,X).\,e'\ C \in [\![\Theta,\beta \vdash [A/\alpha]B\ \text{type}]\!](\theta, X/\beta)$	Definition	(11)

The basic idea is right, but you need more work with variables. Will discuss.

   Fix (C,X) for arbitrary (C,X)

$e\ C \in [\![\Theta,\beta \vdash [A/\alpha]B\ \text{type}]\!](\theta, X/\beta)$	by 11	(12)
$e\ C \in [\![\Theta,\alpha,\beta \vdash B\ \text{type}]\!](\theta, [\![\Theta \vdash A\ \text{type}]\!]\theta, X/\beta)$	Induction	(13)
$\forall(C,X).\,e\ C \in [\![\Theta,\alpha,\beta \vdash B\ \text{type}]\!](\theta, [\![\Theta \vdash A\ \text{type}]\!]\theta, X/\beta)$	by 13	(14)
$e \in [\![\Theta,\alpha \vdash \forall\beta.\,B\ \text{type}]\!](\theta, [\![\Theta \vdash A\ \text{type}]\!]\theta)$	by definition	(15)

   (3) Prove the fundamental lemma for the forall-introduction case $\Theta;\Gamma \vdash \Lambda\alpha.\,e : \forall\alpha.\,A$.

   The Fundamental Lemma states that $\Theta = \alpha_1,\ldots,\alpha_n$, $\Gamma = x_1 : A_1,\ldots,x_n : A_n$ and $\Theta;\Gamma \vdash e : B$, $\Theta \vdash \Gamma$ ctx, θ interprets Θ and for each $x_i : A_i \in \Gamma$, we have $e_1 \in [\![\Theta \vdash A_i\ \text{type}]\!]\theta$ then we have $[C_1/\alpha_1,\ldots C_k/\alpha_k][e_1/x_1 \ldots e_n/x_n]e \in [\![\Theta \vdash B\ \text{type}]\!]\theta$.

We must prove this for the forall-introduction case $\Theta; \Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, A$:

$$\Theta; \Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, A \qquad (16)$$ Assumption

$$\Theta \vdash \Gamma \text{ ctx} \qquad (17)$$ Assumption

$$\theta \text{ interprets } \Theta \qquad (18)$$ Assumption

$$\forall x_i : A_i \in \Gamma.\, e_1 \in [\![\Theta \vdash A_i \text{ type}]\!]\theta \qquad (19)$$ Assumption

$$\Theta; \Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, A \qquad (20)$$ Assumption

$$\Theta, \alpha; \Gamma \vdash e : A \qquad (21)$$ Definition

Consider arbitrary (C, X)

$$(\theta, X/\alpha) \text{ interprets } (\Theta, \alpha) \qquad (22)$$ Definition on 18

$$[C_1/\alpha_1, \ldots, C_k/\alpha_k, C/\alpha][e_1/x_1, \ldots, e_n/x_n]e \in [\![\Theta, \alpha \vdash B \text{ type}]\!](\theta, X/\alpha) \qquad (23)$$ Induction

$$[C_1/\alpha_1, \ldots, C_k/\alpha_k][e_1/x_1, \ldots, e_n/x_n](\Lambda\alpha.\, e)\ C \in [\![\Theta, \alpha \vdash B \text{ type}]\!](\theta, X/\alpha) \qquad (24)$$ Inversion

$$\forall(C, X).\, [C_1/\alpha_1, \ldots, C_k/\alpha_k][e_1/x_1, \ldots, e_n/x_n](\Lambda\alpha.\, e)\ C \in [\![\Theta, \alpha \vdash B \text{ type}]\!](\theta, X/\alpha) \qquad (25)$$ by 24

$$[C_1/\alpha_1, \ldots, C_k/\alpha_k][e_1/x_1, \ldots, e_n/x_n](\Lambda\alpha.\, e) \in [\![\Theta \vdash \forall\alpha.\, B \text{ type}]\!]\theta \qquad (26)$$ Definition of $[\![-]\!]$

✓

I found this diffucult and lack intuition for which operations are permitted i.e. 24 I agree this is tricky.

5. Complete Exercise 1 from Lecture 7. Show how Landin's knot works by unrolling fib(3).

```
let fib = knot (fun f n -> if n <= 1 then 1 else n * f (n - 1))
```

Landin's knot is defined as

```
let knot =
    fun f ->
        let r = ref (fun n -> 0) in
        let recur = fun n -> !r n in
        let () = r := fun n -> f recur n in
        recur
```

By substituting the definition of `fib`, we have:

```
let fib =
    let r = ref (fun n -> 0) in
    let recur = fun n -> !r n in
    let () = r := fun n -> (fun f n -> if n<=1 then 1 else n * f (n-1)) recur n in
    recur
```

By substituting the definition of `fib(3)`, we have:

```
let r = ref (fun n -> 0) in
let recur = fun n -> !r n in
let () = r := fun n -> if n <= 1 then 1 else n * recur (n - 1) in
recur 3
```

By expanding the definitions a few more levels, we have

```
let r = ref (fun n -> 0) in
let recur = fun n -> !r n in
let () = r := fun n -> if n <= 1 then 1 else n * recur (n - 1) in
3 * recur 2
```

Expanding the definitions a few more levels, we have

```
let r = ref (fun n -> 0) in
let recur = fun n -> !r n in
let () = r := fun n -> if n <= 1 then 1 else n * recur (n - 1) in
3 * 2 * 1 * recur 0
```

We see that `recur` is the recursive call. By making a cycle of references using `r`, we get `recur` to indirectly call itself. ✓

6. This question concerns the monadic language given in Lecture 8.

   (a) Given types X and Y define a term:

   $$\cdot;\cdot \vdash \text{fmap} : (X \to Y) \to (T\ X \to T\ Y)$$

   such that for all terms $f : X \to Y$ and values $v : X$ and $v' : Y$ where $fv \rightsquigarrow^* v'$ we have

   $$\langle \sigma, \text{let } y = \text{fmap } f\ \{\text{return } v\}; \text{return } y\rangle \rightsquigarrow^* \langle \sigma, \text{return } v'\rangle$$

   Define fmap as follows:

   $$\text{fmap} = \lambda f : X \to Y.\, \lambda x : X.\, \text{let } z = x; \text{return } (f\ z)$$

   Need { } brackets in the expression I believe.

   (b) For every type X, define terms:

   $$\cdot;\cdot \vdash \eta_X : X \to T\ X$$
   $$\cdot;\cdot \vdash \mu_X : T\ (T\ X) \to T\ X$$

   such that for all values $v : T\ X$ and $v' : X$, where $\langle \sigma, \text{let } x = v; \text{return } x\rangle \rightsquigarrow^* \langle \sigma', \text{return } \sigma'\rangle$ we have

   $$\langle \sigma, \text{let } y = \mu_X(\eta_{(T X)}\ v); \text{return } y\rangle \rightsquigarrow^* \langle \sigma', \text{return } v'\rangle$$

   $$\eta_X = \lambda x : X.\, \text{return } x$$
   $$\mu_X = \lambda x : T\ (T\ X).\, \text{let } z = x; z$$ ✗

   Almost, but the monadic language in this course is stricter.

7. ## 3 2019 Paper 9 Question 14

   (b) In System F, given (i) a Church encoding Nat for the natural numbers, (ii) a Church encoding for the Zero : Nat and Succ : Nat → Nat constructors, and (iii)

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2019p9q14.pdf a type and definition for the iteration operator Iter for natural numbers.

$$\begin{aligned}
\mathsf{Nat} &= \forall\alpha.\, \alpha \to (\alpha \to \alpha) \to \alpha\\
\mathsf{Zero} &= \Lambda\alpha.\, \lambda z.\, \lambda s.\, z\\
\mathsf{Succ} &= \Lambda\alpha.\, \lambda z.\, \lambda s.\, f\ z\\
\text{typeof}(\mathsf{Iter}) &= \forall\alpha.\, \mathsf{Nat} \to \alpha \to (\alpha \to \alpha) \to \alpha\\
\mathsf{Iter} &= \Lambda\alpha.\, \lambda n : \mathsf{Nat}.\, \lambda e_1 : \alpha.\, \lambda e_2 : \alpha \to \alpha.\, n\ \alpha\ e_1\ e_2
\end{aligned}$$ ✓

8. (i) In System F, give a Church encoding for (i) an Option_A type, (ii) the definitions of the $\mathsf{None} : \mathsf{Option}_A$ and $\mathsf{Some} : A \to \mathsf{Option}_A$ operations, and (iii) the type and definition of the case operation on options.

$$\begin{aligned}
\mathsf{Option}_A &= 1 + A = \forall\alpha.\, (1 \to \alpha) \to (A \to \alpha) \to \alpha\\
\mathsf{None} &= \Lambda\alpha.\, \lambda n : \alpha.\, \lambda s : A \to \alpha.\, n\\
\mathsf{Some} &= \lambda x : A \to \Lambda\alpha.\, \lambda n : \alpha.\, \lambda s : A \to \alpha.\, g\ s(x)\\
\text{typeof}(\mathsf{Case}[X]) &= X \to (A \to X) \to (1 + A) \to X\\
\mathsf{Case}[X] &= \lambda n : X.\, \lambda s : A \to X.\, \lambda x : 1 + A.\, x\ X\ n\ s
\end{aligned}$$ ✓

(ii) Assume that $n : B$ and $s : A \to B$, and then

(I) Prove that $\mathsf{Case}[B]\ n\ s\ \mathsf{None} = n$

$$\begin{array}{ll}
\mathsf{Case}[B]\ n\ s\ \mathsf{None} & \\
=(\lambda n' : B.\, \lambda s' : A \to X.\, \lambda x : 1 + A.\, x\ B\ n'\ s')\ n\ s\ \mathsf{None} & \text{Definition of } \mathsf{Case}[B]\\
=\mathsf{None}\ X\ n\ s & \text{Substitution}\\
=(\Lambda\alpha.\, \lambda n : \alpha.\, \lambda s : A \to \alpha.\, n)\ X\ n\ s & \text{Definition of } \mathsf{None}\\
=n \quad ✓ & \text{Substitution}
\end{array}$$

(II) Prove that $\mathsf{Case}[B]\ n\ s\ (\mathsf{Some}\ x) = s\ x$

$$\begin{array}{ll}
\mathsf{Case}[B]\ n\ s\ (\mathsf{Some}\ x) & \\
=(\lambda n' : B.\, \lambda s' : A \to X.\, \lambda x : 1 + A.\, x\ B\ n'\ s')\ n\ s\ (\mathsf{Some}\ x) & \text{Definition of } \mathsf{Case}[B]\\
=\mathsf{Some}\ x\ X\ n\ s & \text{Substitution}\\
=(\lambda x : A \to \Lambda\alpha.\, \lambda n : \alpha.\, \lambda s : A \to \alpha.\, g\ s(x))\ x\ X\ n\ s & \text{Definition of } \mathsf{Some}\\
=s\ x \quad ✓ & \text{Substitution}
\end{array}$$

(iii) In System F, define a predecessor operation $\mathsf{Pred} : \mathsf{Nat} \to \mathsf{Nat}$, which returns Zero if given Zero as an argument, and returns n if given $\mathsf{Succ}\ n$ as an argument.

$$\mathsf{Pred} = \lambda n : \mathsf{Nat}.\, \mathsf{Case}[\mathsf{Nat}]\ (n\ \mathsf{None}\ (\mathsf{Case}[1{+}\mathsf{Nat}]\ (\mathsf{Some}\ \mathsf{Zero})\ (\lambda m.\, \mathsf{Some}\ (\mathsf{Succ}\ n))))\ 0\ (\lambda n.\, n)$$ ✓

(iv) In System F, define a subtraction operator $\mathsf{sub} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$ which is defined to be *saturating*. That is, $\mathsf{sub}\ m\ n$ returns the difference if $m \geq n$ and returns 0 otherwise.

$$\mathsf{sub} = \lambda m : \mathsf{Nat}.\, \lambda n : \mathsf{Nat}.\, \mathsf{Iter}(n, z \to m, s(x) \to \mathsf{Pred}\ x)$$ ✓

9. 4 2021 Paper 9 Question 15

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2021p9q15.pdf

(a) In a simply-typed lambda calculus augmented with first-class continuations, booleans, a list type and its iterator (i.e., fold, but not full recursion), write a function

$$\mathsf{every} : (X \to \mathsf{Bool}) \to \mathsf{List}\ X \to \mathsf{Bool}$$

such that every p xs returns true if every element of xs satisfies p, and false otherwise. This function should also stop iterating over the list as soon as it finds a false element. You may use SML or OCaml-style notation if desired, but explain any notation beyond the basic lambda-calculus.

I was confused by "stop iterating over the list as soon as it finds a false element": as I understand it, this seems impossible without recursion (as fold is atomic).

The interpretation which (by elimination) I settled on was that the *predicate* would not be unnecessarily evaluated on the elements of the list once the result of computation was known, but that fold itself could continue.

every $= \lambda p : \mathsf{Bool} \to X.\, \lambda xs : \mathsf{List}\ X.\, \mathsf{fold\ true}\ (\lambda acc.\, \lambda x : X.\, \text{if } acc \text{ then } p\ x \text{ else } acc)$

Not quite, you can use exceptions to terminate early.

(b) In the monadic lambda calculus with stat, suppose we change the typing rule for reading locations to not cause a monadic effect: if we suggest changing the monadic lambda calculus to permit treating reads as pure:

$$\frac{\ell : X \in \Sigma}{\Sigma; \Gamma \vdash \ell : X}$$

i. Is this rule still typesafe? Informally but carefully justify your answer.

No. Consider Landin's knot. We can "tie" Landins knot and the return value is an expression which only reads from locations. This expression will loop infinitely and so is not typesafe. However, if reads are pure then it *will* be considered as typesafe. So this rule is **not** typesafe.

Not quite. The expression is still typesafe. Progress and preservation aren't violated. Termination isn't necessary for type safety.

ii. Is the following *common subexpression elimination* transformation sound? Either give an argument why it is, or supply a counterexample and explain why it shows it is not.

```
let x = return e1;                  let x = return e1;
let y = e2;             =====>      let y = e2;
let z = return e1;                  [z/x]e3
e3
```

I presume the original expression was typo'd and meant to read `[x/z]e3` – else the second expression can be badly-scoped by any term e_3 which uses either z or x.

That's right.

This transformation is **not** safe. Consider the situation that e_1 is an impure computation (with non-idempotent side effects) and e_3 is some expression whose result depends on the state that is changed by e_1. We would have that the left-hand expression would run e_1 twice while the one on the right would only run it once. This means that they would start executing e_3 with different stores!

Example: $e_1 = \ell :=!\ell + 1$, $e_2 = ()$, $e_3 = !\ell$ with the LHS expression denoted by *LHS* and the RHS expression denoted by *RHS*. $\langle \ell : 0, \mathit{LHS}\rangle \rightsquigarrow \langle \ell : 2, 2\rangle$ while $\langle \ell : 0, \mathit{RHS}\rangle \rightsquigarrow \langle \ell : 1, 1\rangle$. Clearly, we can see that these expressions are **not** the same.

✓

10. 5 2020 Paper 9 Question 15

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2020p9q15.pdf

(b) Suppose we extend the simply-typed lambda calculus with the ability to raise exceptions with the fail construct, and the ability to catch exceptions with the try e_1 except e_2 construct. Suppose also that we track exceptions monadically, with the type Exn A representing possibly-failing computations of A.

(i) Give a typing rule for signalling an error with fail.

$$\frac{}{\Gamma \vdash \mathsf{fail} : \mathsf{Exn}\ A}$$ ✓

(ii) Give a typing rule for trapping an error with try e_0 except e_1. Does your type for this term have an effect? Justify your design.

$$\frac{\Gamma \vdash e_0 : \mathsf{Exn}\ A \qquad \Gamma \vdash e_1 : A}{\Gamma \vdash \mathsf{try}\ e_0\ \mathsf{except}\ e_1 : A}$$

This term can have an effect: consider the case $A = \mathsf{Exn}\ B$ for some B. In this case, we have that the return value from this term is *still* itself an exception. This means that it affects control flow of other terms and thus has an effect! ✓

6 Extension Questions

1. Write a term `and` that takes two Church booleans and returns their conjunction. (How do you change this for `or`?)

$$\mathtt{and} = \Lambda\beta.\,\lambda b_1 : \forall\alpha.\,\alpha \to \alpha \to \alpha.\,\lambda b_2 : \forall\alpha.\,\alpha \to \alpha \to \alpha.\,\lambda t : \beta.\,\lambda f : \beta.\,b_1\ \alpha\ (b_1\ \alpha\ t\ f)\ f$$ ✓

To change for `or`, we shift the second boolean to be the result of the false condition.

$$\mathtt{or} = \Lambda\beta.\,\lambda b_1 : \forall\alpha.\,\alpha \to \alpha \to \alpha.\,\lambda b_2 : \forall\alpha.\,\alpha \to \alpha \to \alpha.\,\lambda t : \beta.\,\lambda f : \beta.\,b_1\ \alpha\ t\ (b_1\ \alpha\ t\ f)$$ ✓

2. Write a function `equal` that takes two Church numerals and returns a church boolean.

Using the sub and and functions defined earlier,

$$\mathtt{equal} = \lambda n : \mathsf{nat}.\,\lambda m : \mathsf{nat}.\,\mathsf{and}(\mathsf{iter}(\mathsf{sub}(m,n), z \to \mathsf{true}, s(x) \to \mathsf{false}), \mathsf{iter}(\mathsf{sub}(n,m), z \to \mathsf{true}, s(x) \to \mathsf{false}))$$ ✓

3. 7 2014 Paper 9 Question 13

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2014p9q13.pdf

(b) Find, with justification, a PLC type τ for which the following typings are both provable:

$$\{\} \vdash \Lambda\alpha.\,\Lambda\beta.\,\lambda x : \alpha.\,\lambda y : \beta.\,\Lambda\gamma.\,\lambda z : \tau.\,z\ x\ y : \forall\alpha.\,\forall\beta.\,\alpha \to \beta \to \forall\gamma.\,(\tau \to \gamma)$$

$$\{\} \vdash \Lambda\alpha.\,\Lambda\beta.\,\lambda z : \forall\gamma.\,\tau \to \gamma.\,z\ \alpha\ (\lambda x : \alpha.\,\lambda y : \beta.\,x) : \forall\alpha.\,\forall\beta.\,(\forall\gamma.\,\tau \to \gamma) \to \alpha$$

$$\tau = \alpha \to \beta \to \gamma$$

In the first expression, we have $z : \tau$. We see that z is passed $x : \alpha$ and $y : \beta$ as arguments and returns γ. Therefore in the first expression, $\tau = \alpha \to \beta \to \gamma$.

In the second expression, we have $z : \forall\gamma.\,\tau \to \gamma$. We then see $z\ \alpha\ (\lambda x : \alpha.\,\lambda y : \beta.\,x) : \alpha$. So we have $\tau = \alpha \to \beta \to \alpha$. But, since α was passed to z as argument, α and γ were unified; so we can replace any occurrence of an α in the type of z with a γ and the type will still be correct. So $\tau = \alpha \to \beta \to \gamma$ is a possible type. ✓ This is the only such which is consistent with the first equation.

(c) Give infinitely may different closed PLC expressions in beta-normal form of type $\forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)$.

Consider expressions of the form:

$$\Lambda\alpha.\,\lambda f : \alpha \to \alpha.\,\lambda x : \alpha.\, f^n(x)$$

where $f^n(x) = \underbrace{f(\ldots(f}_{\times n}(x)\ldots)$. ✓

(d) Ues your answer to part (a) to show that there is no closed PLC expression Y of type $\forall\alpha.\,(\alpha \to \alpha) \to \alpha$ for which the beta-reduction $Y\ \alpha\ f \to f(Y\ \alpha\ f)$ **holds**.

Assume there is such a Y. By definition, we have Y nat $\mathsf{succ} \rightsquigarrow \mathsf{succ}(Y$ nat $\mathsf{succ})$. Since all terms in PLC terminate, we also have Y nat $\mathsf{succ} \rightsquigarrow^* n$ for some n. Thus, we have $n = n + 1$. This is clearly false and therefore the original assumption that some Y existed which had this condition must be false. Therefore, no such Y can exist. ✓

Good work!