PoPL Spring 2026, Final Review 1

EN.601.426/626 Principles of Programming Languages

Name: _____

1. **Subtyping Between Int and Bool.** Normally in TF$^\flat$, we would not expect that `Int` and `Bool` are in a subtyping relationship, since they are different types and are disjoint sets. However, in practical programming languages like C and Python, we can treat `Int` and `Bool` as being in a subtyping relationship, where `Bool` <: `Int`. Before talking about the operational semantics, let's first see if we can make the type system work:

1.1. Please see if the following expression can be typed, and if so, what is its type? Draw out the proof tree for the typing judgement.

$$\frac{}{\vdash \texttt{(Fun x: Int -> x + 1) True} :}$$

1.2. For a program that type checks, we would like it to not get stuck during evaluation. However, under normal operational semantics of TF$^\flat$, the expression above in 1.1 will get stuck, since we cannot apply `+` to a boolean. Please write the new operational semantics rules for `+` and `-` so that we can evaluate the expression.

1.3. With this subtyping relationship, what is the relationship between the following types? Are they subtype, supertype, or neither?

`Bool -> Bool` vs. `Int -> Int`

`Bool -> Int` vs. `Int -> Int`

`Int -> Bool` vs. `Int -> Int`

1.4. Alternatively (without `Bool` <: `Int`), let us imagine a world where we have `Int` <: `Bool`. Please come up with a program that can type check under this new world but cannot type check under the original $TF^\flat$.

2. **Optional and Unwrap.** In this question, we will work with F♭-Optional, which includes an extension of F♭ with an optional type. We extend the syntax of F♭ with the following constructs:

$$\begin{array}{rcl} e & ::= & (\ldots \text{F}^\flat \text{ Exprs} \ldots) \mid \texttt{None} \mid \texttt{Some}(e) \mid \texttt{Unwrap}\ e\ \texttt{Default}\ e \\ v & ::= & (\ldots \text{F}^\flat \text{ Values} \ldots) \mid \texttt{None} \mid \texttt{Some}(v) \end{array}$$

The operational semantics of `None` and `Some` are straightforward, just like OCaml's `option` type. The meaning of `Unwrap` should be clear from a couple examples:

$$\begin{array}{rl} \texttt{Unwrap Some(1) Default 5} & \Rightarrow \texttt{1} \\ \texttt{Unwrap None Default 5} & \Rightarrow \texttt{5} \\ \texttt{Unwrap 33 Default 5} & \Rightarrow \text{(stuck)} \\ \texttt{Unwrap Some(0) Default (0 0)} & \Rightarrow 0 \end{array}$$

2.1. Write the operational semantics rule for the three new constructs.

2.2. Please evaluate the following expression and draw the proof tree.

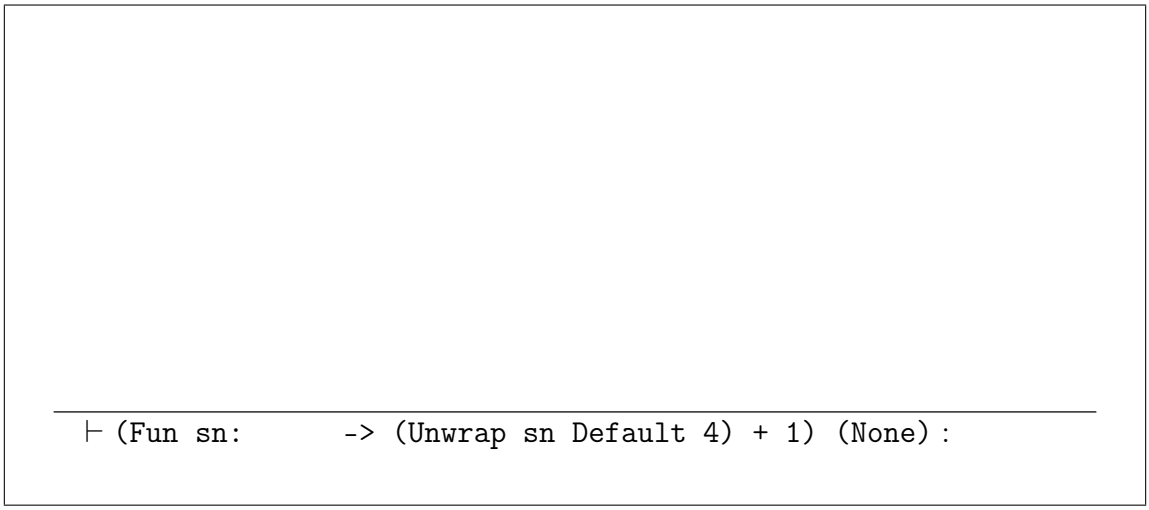
$$\frac{}{\texttt{(Fun sn -> (Unwrap sn Default 4) + 1) (None)} \Rightarrow}$$

2.3. We would like to extend our type system to support the new constructs. Specifically, our type system is extended with the following type:

$$t \quad ::= \quad (\ldots \mathrm{TF}^{\flat}\ \mathrm{Types} \ldots) \mid \texttt{Option<}t\texttt{>}$$

Please write the typing rules $\Gamma \vdash e : t$ for the three new constructs.

2.4. Please add type annotation to the expression in 2.2, infer its type, and draw the proof tree for the typing judgement.



3. **State and Operational Equivalence.** Recall that operational equivalence for $F^\flat$ is defined on that two expressions either both diverge or both evaluate to the same value under any evaluation context C.

However, for $F^\flat S$, which is $F^\flat$ with state, we need to consider the state as well. We say that $e_1 \cong e_2$ if for any evaluation context C and any initial state σ, we have:

$$\langle C[e_1], \sigma \rangle \Rightarrow \langle v_1, \sigma_1 \rangle, \text{ and } \langle C[e_2], \sigma \rangle \Rightarrow \langle v_2, \sigma_2 \rangle$$

and that $v_1 \cong v_2$ and $\sigma_1 = \sigma_2$. When v_1 and v_2 are simple values (booleans, integers, etc.), we say that $v_1 \cong v_2$ if they are the same value. When v_1 and v_2 are functions, we need to further check that for any argument, they are still operationally equivalent.

3.1. Please determine whether these two expressions are operationally equivalent:

$e_1 =$ `Fun x -> (x := 1; !x)` $\qquad$ $e_2 =$ `Fun x -> (x := 1; 1)`

3.2. Please determine whether these two expressions are operationally equivalent.

$$e_1 = \left\{ \begin{array}{l} \texttt{temp := !x;} \\ \texttt{x := !y;} \\ \texttt{y := !temp;} \\ \texttt{x} \end{array} \right\} \qquad e_2 = \left\{ \begin{array}{l} \texttt{temp := !y;} \\ \texttt{y := !x;} \\ \texttt{x := !temp;} \\ \texttt{x} \end{array} \right\}$$

3.3. In the definition, we require that $\sigma_1 = \sigma_2$. Note that the state σ is a mapping from cell names to values. For two σ-s to be equal, that means we need the cell names to be exactly the same as well. With this assumption, are the following two expressions operationally equivalent? The answer is No!

Give an example context C and two proof trees of evaluation for $C[e_1]$ and $C[e_2]$ that show that they are not operationally equivalent.

$e_1 =$ `Let x = Ref 0 In x` $\qquad$ $e_2 =$ `Let x = Ref 0 In x`

This is bad! The two expressions are syntactically the same, so they *should* be operationally equivalent.

3.4. To fix this problem, we can relax the definition of operational equivalence of $e_1 \cong e_2$ by requiring that $\sigma_1 \cong \sigma_2$ (other parts stay the same), which means that they are equal if they have the same values, regardless of the cell names. This is called *location invariance*. Can you write out the formal definition of $\sigma_1 \cong \sigma_2$ accounting for locational invariance?

$e_1 \cong e_2 \iff$

4. **Actors That Can Exit.** In the actor models, actors will keep existing in the global system and is never ending—whatever is returned as a continuation will be kept, even if it is never used again. However, we would like an actor to be able to exit on itself, similar to how a C program can call `exit(0)` to exit itself.

We are going to extend the global operational semantics $G \to G'$ with a new rule for actors to exit. Specifically, when an actor a is handling a message, if the returned continuation is `'Exit`, then the actor will be removed from the global system.

Note that the *original* global layer operational semantics of AF$^\flat$V is

$$\frac{h_a\ m \overset{S}{\Rightarrow} c_a}{G \cup \{\langle a, h_a\rangle, [a \leftarrow m]\} \to G \cup S \cup \{\langle a, c_a\rangle\}}$$

That is, an actor a with handler h_a is handling a message $[a \leftarrow m]$ and returns a continuation c_a. That c_a is then added to the global system as the new handler of actor a, along with the side-effects S.

4.1. Please now write the new global operational semantics rule for actors to exit.

4.2. Given the current global state G (note that `self` is referring to the current actor's message handler given by, presumably, a Y-combinator):

$$G = \{\langle a, h_a\rangle, [a \leftarrow 1], [a \leftarrow 2]\}$$
$$h_a = \texttt{Fun msg -> If msg = 1 Then 'Exit Else self}$$

What are all possible next global states after G?