Day 5.

1. Eagerness

Let's revisit our evaluation rule for `let`

$$\frac{e_1 \Downarrow v_1 \quad e_2[v_1/x] \Downarrow v_2}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2}$$

- Do we have to evaluate e_1 *before* substituting?
- What does this tell us about (apparently degenerate) terms like $\texttt{let}\ x = \texttt{if}\ 1\ \texttt{then}\ 2\ \texttt{else}\ 3\ \texttt{in}\ 5$? What's our intuition for what this term *should* mean?
- What does this tell us about terms like $\texttt{let}\ x = 5 \times 5\ \texttt{in}\ x \times x$? How much work *should* this term do?

An alternate approach: evaluate *after* substituting:

$$\frac{e_2[e_1/x] \Downarrow v}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v}$$

How does this effect our definition of substitution?

- We already defined substitution for expressions, relying on the inclusion $\mathcal{V} \in \mathcal{E}$, so substituting non-value terms doesn't cause any problems.
- Our definition of substitution for `let` doesn't have to change:

  $$(t_2[t_1/y])[t/x] \approx (t_2[t/x])[t_1[t_2/x]/y]$$

  (modulo usual tedious side conditions on variables appearing in t_1 and t_2.

Nomenclature (derived from Algol 68). Note that these issues appear identically when we start talking about functions, ergo "call-by-X".

- Evaluating *before* substituting is called *call-by-value.* Name here is relatively intuitive: by *value* because the thing being substituted is a value. More predictable performance, but more complex equations.
- Evaluating *after* substituting is called *call-by-name.* Name here is less intuitive, but think of passing around *names* of terms rather than their values. *This is not pass-by-reference... still no mutation to hand.* Simpler equational theory, but less predictable performance.

Each approach can leak into the other:

- *Futures* in modern programming languages give a flavor of call-by-name in a call-by-value language—the future itself doesn't contain the value, but rather a promise that the value will someday be computed.
- *Call-by-need* in Haskell moderates the cost of call-by-name reduction, by only evaluating each term once even if the term seems to have been copied.

2. Environments

We can attempt to follow our existing approach to approximate the behavior of `let`. However, a problem emerges. Consider the type system we've built in the past. If we try to extend it to `let`, we get something like:

$$\frac{e_1 : t_1 \quad e_2[??/x] : t_2}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2}$$

but what to put in for ??? We can't substitute types into terms—while we had $\mathcal{V} \subseteq \mathcal{E}$, we certainly don't have $\mathcal{T} \subseteq \mathcal{E}$.

An aside. It might seem like the call-by-name let rule gives us hope: why can't we have:

$$\frac{t_2[t_1/x] \Downarrow_{\pm} s}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \Downarrow_{\pm} s}$$

There are two reasons. First, this isn't very approximate—we're approximating the value of t_1 once for each time x appears in t_2. Second, and more important, this doesn't work for recursion... which we haven't talked about yet, but we will.