Day 9

1. Functions and Environments

To develop our approximation of local variables, we needed to move from a substitution-based view of evaluation to an environment-based view. We’ll have to do something similar for functions. So, let’s get started!

\[
H, x \Downarrow cbv H(x) \quad H, \lambda x.t \Downarrow cbv \lambda x.t
\]

- Omitted rules for numeric constants, because they don’t behave any different than they did in the last version
- Again, reusing syntax for 2-place and 3-place evaluation relations

We should confirm that it works. Let’s try some simple reductions:

\[
\emptyset, \lambda a.\lambda b.b \Downarrow \lambda a.\lambda b.b \\
\emptyset, 3 \Downarrow 3 \quad \{a \mapsto 3\}, \lambda b.b \Downarrow \lambda b.b
\]

\[
\emptyset, (\lambda a.\lambda b.b)3 \Downarrow \lambda b.b \\
\emptyset, 2 \Downarrow 2 \\
\emptyset, (\lambda a.\lambda b.b)3 \Downarrow 2 \Downarrow 2
\]

What’s gone wrong?

- We’re trying to use variable \(a\) when it’s not apparently in scope. Fair enough—this shouldn’t be derivable.
- Variable \(a\) should have gotten its meaning in reducing the left-hand argument, but it didn’t. This is the real problem.
- Missing one aspect of substitution—although evaluation doesn’t touch \(\lambda\)s, substitution does!

Solution: \(\lambda\) terms need to carry their defining environments with them!

- Means we don’t have to reintroduce substitution
- Combination of a function and its environment called a closure.

2. Closures

Let’s recap our language:
\[
\begin{align*}
\mathcal{X} & \ni x \\
\mathcal{V} & \ni v ::= z \mid \langle H, \lambda x.t \rangle \\
\mathcal{T} & \ni t ::= z \mid t_1 \circ t_2 \mid x \mid \lambda x.t \mid t_1 t_2
\end{align*}
\]

- New value form: closures. Package environment with function
  - Values no longer subset of terms... but can think of \( \langle H, \lambda x.t \rangle \) as being syntax for \((\lambda x.t)[v_i/y_i]\) where \(H = \{y_i \mapsto v_i\}\).

Now we can adjust evaluation rules to construct and use closures.

\[
\begin{array}{c}
H, \lambda x.t \Downarrow \langle H, \lambda x.t \rangle \\
H, t_1 \Downarrow \langle H', \lambda x.t \rangle \\
H, t_2 \Downarrow w \\
H'[x \mapsto w], t \Downarrow v
\end{array}
\]

Does this work?

\[
\begin{array}{c}
\emptyset, \lambda a.\lambda b.b \Downarrow \lambda a.\lambda b.b \\
\emptyset, 3 \Downarrow 3 \\
\emptyset, (\lambda a.\lambda b.b) 3 \Downarrow \{a \mapsto 3\}, \lambda b.b
\end{array}
\]

Looks promising.

\[
\begin{array}{c}
\emptyset, (\lambda a.\lambda b.a) \Downarrow \langle \emptyset, \lambda a.\lambda b.a \rangle \\
\emptyset, 3 \Downarrow 3 \\
\emptyset, (\lambda a.\lambda b.a) 3 \Downarrow \{a \mapsto 3\}, \lambda b.a
\end{array}
\]

Seems to work!

Call by name variation: just replace \(H \in \mathcal{X} \rightarrow \mathcal{V}\) with \(H \in \mathcal{X} \rightarrow \mathcal{T}\) and:

\[
\begin{array}{c}
H, x \Downarrow \langle H', \lambda x.t \rangle \\
H, t \Downarrow v \quad \langle H', \lambda x.t \rangle \quad H'[x \mapsto t_2], t \Downarrow v
\end{array}
\]

**Historical note.** Early implementations of LISP, including some still in use (ELISP), got closures wrong. Some people like to present this as a design choice; they call it “dynamic scope” or similar euphemisms. This is not a design choice, any more than \(2 + 2 = 5\) would be a design choice for addition. It is a system that fails to match the semantics of the \(\lambda\)-calculus.

### 3. Typing Functions

What can go wrong? \(1 2, (\lambda c.c) + 1\).

We need to extend our grammar of types:

\[
\begin{align*}
\mathcal{V} & \ni T ::= \text{Int} \mid T_1 \rightarrow T_2
\end{align*}
\]

- Why don’t closures need to be reflected in the types of functions?
As before, we define a variation of the evaluation relation that characterizes the types of values: \( \Gamma \vdash t : T \).

- Syntax: \( \vdash \) denotes consequence—under the assumptions in \( \Gamma \), the typing on the right holds.
- \( \Gamma : \mathcal{X} \rightarrow \mathcal{Y} \) maps from variables to their types.
- More about the typing relation... and the significance of our notational choices... to come.

Typing rules:

\[
\begin{align*}
\Gamma \vdash z : \text{Int} & \quad \frac{\Gamma \vdash t_1 : \text{Int} \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash t_1 + t_2 : \text{Int}} \\
\Gamma \vdash x : \Gamma(x) & \quad \frac{\Gamma \vdash \lambda x.t : T_1 \rightarrow T_2 \quad \Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 \ t_2 : T_2}
\end{align*}
\]

- Common notation for \( \Gamma[x \mapsto T_1] \) is \( \Gamma, x : T_1 \). May fall into this later, but not yet.
- Why don’t we have to represent the closure in the application rule?

Let’s look at some simple derivations:

\[
\begin{align*}
\{a \mapsto \text{Int}, b \mapsto \text{Int} \rightarrow \text{Int}\} & \vdash a : \text{Int} \\
\{a \mapsto \text{Int}\} & \vdash \lambda b.a : (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \\
\emptyset & \vdash (\lambda a.\lambda b.a) : \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \\
\emptyset & \vdash 3 : \text{Int} \\
\emptyset & \vdash \lambda c.c : \text{Int} \rightarrow \text{Int} \\
\emptyset & \vdash (\lambda a.\lambda b.a) \ 3 \ (\lambda c.c) : \text{Int} \\
\end{align*}
\]

- Check typing of functions at construction, not at use. So: more structure under the typing of a \( \lambda \), but less at their uses.
- Same term may have more than one typing derivation: \( \lambda a.a \) (up to \( \alpha \)-equivalence) given both \( \text{Int} \rightarrow \text{Int} \) and \( (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int}) \).