Day 7.

1. Functions

Our next new language feature is *functions*. The good news is, we've already built up much of the machinery we'll need when talking about local variables.

We'll start with the terms of our language:

$$\begin{aligned}\mathcal{X} &\ni x \\ \mathcal{V} &\ni v ::= z \mid \lambda x.t \\ \mathcal{E} &\ni t ::= z \mid t_1 \odot t_2 \mid x \mid \lambda x.t \mid t_1\ t_2\end{aligned}$$

We let $\odot$ stand in for the usual collection of binary arithmetic operations; they're not going to contribute a great deal from this point in the course onwards, so we'll rarely pay them much attention. We have two new term forms: function *abstractions* $\lambda x.t$ and function *applications* $t_1\ t_2$. Functions themselves are also values, so we have a new value form $\lambda x.t$.

- Abstractions extend as far to the right as possible. The term $\lambda x.x\ y$ is interpreted as $\lambda x.(x\ y)$, not as $(\lambda x.x)\ y$.
- Application binds as tightly as possible. The term $fx + y$ is interpreted as $(f\ x) + y$, not as $f\ (x + y)$.
- Application associates to the left. The term $f\ x\ y$ is interpreted as $(f\ x)\ y)$, not as $f\ (x\ y)$.
- We don't *need* `let` any more: the term `let` $x = t_1$ `in` t_2 is equivalent to $(\lambda x.t_2)\ t_1$. Nevertheless, for ease of reading, we'll continue to write `let` on occasion.

We can define substitution for our new language:

$$\begin{aligned} z[t/x] &= z \\ (t_1 \odot t_2)[t/x] &= t_1[t/x] \odot t_2[t/x] \\ (t_1\ t_2)[t/x] &= t_1[t/x]\ t_2[t/x] \end{aligned} \qquad y[t/x] = \begin{cases} t & \text{if } x = y \\ y & \text{otherwise} \end{cases}$$

$$(\lambda y.t_1)[t/x] = \begin{cases} \lambda y.t_1 & \text{if } x = y \\ \lambda y.t_1[t/x] & \text{otherwise} \end{cases}$$

Observe that our old rule for substitution on `let` falls out of the new rule and the encoding for `let` given above.

Similarly, we can define evaluation:

$$\frac{}{z \Downarrow z} \qquad \frac{t_1 \Downarrow v_2 \quad t_2 \Downarrow v_2}{t_1 \odot t_2 \Downarrow v_1 \odot v_2} \qquad \frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \frac{t_1 \Downarrow_{\mathsf{cbv}} \lambda x.t \quad t_2 \Downarrow_{\mathsf{cbv}} w \quad t[w/x] \Downarrow_{\mathsf{cbv}} v}{t_1\ t_2 \Downarrow_{\mathsf{cbv}} v}$$

Aside. In class, I also discussed a version of this rule that attempted to separate out the evaluation of t_1 and t_2 from the substitution itself. The goal was to highlight the computational power of the evaluation rule, not just its administrative details. However, I don't think that actually works very

well in the evaluation framework we're using in this course, so I'm going to ignore it going forward and only use the version of evaluation given here.

Let's consider some simple derivations:

$$\dfrac{\dfrac{}{\lambda x.x + x \Downarrow \lambda x.x + x} \quad \dfrac{\dfrac{}{4 \Downarrow 4} \quad \dfrac{}{3 \Downarrow 3}}{4 + 3 \Downarrow 7} \quad \dfrac{\dfrac{}{7 \Downarrow 7} \quad \dfrac{}{7 \Downarrow 7}}{(a + a)[7/x] \Downarrow 14}}{(\lambda a.a + a)\,(4 + 3) \Downarrow_{\mathsf{cbv}} 14}$$

$$\dfrac{\dfrac{}{\lambda a.3 \Downarrow \lambda a.3} \quad \dfrac{\dfrac{}{4 \Downarrow 4} \quad \dfrac{}{3 \Downarrow 3}}{4 + 3 \Downarrow 7} \quad \dfrac{}{3[7/a] \Downarrow 3}}{(\lambda a.3)\,(4 + 3) \Downarrow_{\mathsf{cbv}} 3}$$

2. Currying

So far, we only have single argument functions, and the first hypothesis of our evaluation derivations has always been a constant reduction. We can represent multiple argument functions using nested single argument functions. For example:

$$\dfrac{\dfrac{\dfrac{}{\lambda a.\lambda b.a \Downarrow \lambda a.\lambda b.a} \quad \dfrac{}{3 \Downarrow 3} \quad \dfrac{}{(\lambda b.a)[3/a] \Downarrow \lambda b.3}}{(\lambda a.\lambda b.x)\,3 \Downarrow \lambda b.3} \quad \dfrac{}{2 \Downarrow 2} \quad \dfrac{}{3[2/b] \Downarrow 3}}{(\lambda a.\lambda b.a)\,3\,2 \Downarrow_{\mathsf{cbv}} 3}$$

- Remember grouping of function application: $(\lambda a.\lambda b.a)\,3\,2$ should be read as $((\lambda a.(\lambda b.a))\,3)\,2$.
- History: named after Haskell B. Curry... except originally introduced 40 years earlier by Moses Shönfinkel. Somehow the alternate name Shönfinkelization never caught on.

3. Non-termination

Our language to this point seems quite simple—what does function abstraction and application really buy you? Here's a simple example: what's the result of evaluating the following term: $(\lambda a.a\ a)\,(\lambda a.a\ a)$. After the first substitution $(a\ a)[\lambda a.a\ a/a]$ we get $(\lambda a.a\ a)\,(\lambda a.a\ a)$ again... and so forth. So our little language is sufficient to represent non-termination at least.

In fact, the Church-Turing Thesis states that (modulo some philosophical wibbling) any computation representable using Turing machines is representable using just variables, abstractions, and applications, and vice versa! These representations may not be entirely intuitive at first, but they provide a powerful, compositional basis for thinking about programming (functional and otherwise). I discuss this more in EECS 762.

That being said, the simple type systems we develop in this course will restrict the λ-calculus such that many of those encoding will be untypable for us. This means that we'll continue to study new ideas as new language features, rarely as being encoded in our existing language. This isn't actually as much of a weakness as it sounds: it'll make it easier to draw connections between λ-calculus and other areas of logic, and can motivate/explain the encodings.

4. Call by Name

Call-by-name evaluation again switches the order of substitution and evaluation:

$$\frac{t_1 \Downarrow_{\mathsf{cbn}} \lambda x.t \quad t[t_2/x] \Downarrow_{\mathsf{cbn}} v}{t_1\ t_2 \Downarrow_{\mathsf{cbn}} v}$$

As we expect, call by name reduction means that unused function arguments don't get evaluated:

$$\frac{\overline{\lambda a.3 \Downarrow \lambda a.3} \quad \overline{3[(\lambda b.b\ b)\ (\lambda b.b\ b)/a] \Downarrow 3}}{(\lambda a.3)\ ((\lambda b.b\ b)\ (\lambda b.b\ b)) \Downarrow_{\mathsf{cbn}} 3}$$