Day 12.

1. Purity

The language we've been studying so far is *pure*:

- Pure *functions*—result of the function determined solely by its input... e.g., sin, $x \mapsto x + 1$
- Pure *computations*—results of the computation determined solely by the value of its free variables... e.g., $\sin x$, $x + 1$
- Pure *language*—only describes pure computations

Purity is useful in understanding and writing correct programs:

- Avoids interaction between different parts of programs
- Useful in concurrency and parallelism in particular
- Related to e.g. `const`-correctness in C++

Impurity isn't *necessary* in languages:

- Haskell, Idris, Agda, etc.

but it is a feature of most programming languages. Regardless, we will need some way to *model* side effects, so we can talk about (e.g.) I/O even in pure languages.

Goals for the next part of the semester:

1. Develop *models* of various side-effects in terms of our evaluation relation.
   - Our language will grow impure terms—that is, terms whose behavior is determined by more than the values of their free variables. However, evaluation itself will remain a (pure) mathematical relation. That's why I call this modeling side-effects.
2. Discover a common structure in all of these models of side-effects—*the monad*—allowing us to generalize our implementations of side-effects
   - This is how Haskell (and other such languages) account for side-effects in a pure language
   - We'll need to learn some details about how Haskell handles generalization—called *parametric* and *qualified* polymorphism—to realize this common structure in our implementations.
3. Design *type and effect* systems that capture side-effect behavior in types.
   - We'll use this as a vehicle to talk about *subtyping*; the latter has wide-ranging importance beyond effect systems.

Sorts of side-effects

- Reader—access to ambient data (i.e., operating system environment, hardware parameters)
- Writer—producing values *in addition to* results (i.e., logging)
- Exceptions—producing values *instead of* results (i.e., errors, failures)
- State—something like interacting reader and writer
- Non-determinism—producing *multiple* results. (In particular, probabilistic programming produces a *distribution* of results, and is one useful way to talk about a variety of machine learning

problems.)
- Concurrency—multiple communicating threads of computation. (*Doesn't* imply parallelism.)
- Continuations—a "general-purpose" side effect capable of implementing most others. (No good intuition)

Things we won't consider side-effects (but some might):

- Non-termination
- Parallelism/computation time

2. Reader

- Idea: terms have access to some ambient, read-only state.
- Operations:
  - Read the ambient state
  - Run a computation with a *different* ambient state

Let's extend our term language with these operations:

$$t ::= z \mid t \odot t \mid x \mid \lambda x.t \mid t\ t \mid \mathsf{ask} \mid \mathsf{local}\ t\ t$$

We'll assume that the ambient state is one of our values—usually an integer, for ease, but could be anything. Values *don't* include side-effecting terms:

$$v ::= z \mid \lambda x.t$$

but note that values may *delay* side effects.

Evaluation relation needs to be extended with this ambient value r. We'll write $t \mid r \Downarrow v$ to denote that term t in ambient state r evaluates to v.

The rules for the new terms are "obvious":

$$\frac{}{\mathsf{ask} \mid r \Downarrow r} \qquad \frac{t \mid r' \Downarrow v}{\mathsf{local}\ r'\ t \mid r \Downarrow v}$$

but that's not enough! We also have to account for the remaining constructs of our language in this new evaluation rule. Let's start with the call-by-value version:

$$\frac{}{z \mid r \Downarrow z} \quad \frac{t_1 \mid r \Downarrow z_1 \quad t_2 \mid r \Downarrow z_2}{t_1 \odot t_2 \mid r \Downarrow z_1 \odot z_2} \quad \frac{}{\lambda x.t \mid r \Downarrow \lambda x.t} \quad \frac{t_1 \mid r \Downarrow \lambda x.t \quad t_2 \mid r \Downarrow w \quad t[w/x] \mid r \Downarrow v}{t_1\ t_2 \mid r \Downarrow v}$$

where ("predictably"):

$$\mathsf{ask}[v/x] = \mathsf{ask} \qquad (\mathsf{local}\ r\ t)[v/x] = \mathsf{local}\ (r[v/x])\ t[v/x]$$

Some patterns emerge:

- New terms interact with new portions of the evaluation relation
- Meaning of old terms stays "relatively" constant... they preserve the ambient state, but don't interact with it
- Using call-by-value function calls (more on this shortly)

12.

Some examples.

$$\frac{\frac{}{\mathsf{ask} \mid 1 \Downarrow 1} \quad \frac{}{1 \mid 1 \Downarrow 1}}{\mathsf{ask} + 1 \mid 1 \Downarrow 2} \qquad \frac{\frac{}{\mathsf{ask} \mid 14 \Downarrow 14} \quad \frac{}{1 \mid 14 \Downarrow 1}}{\mathsf{ask} + 1 \mid 14 \Downarrow 15}$$

- Just knowing the term is no longer enough to determine the result
- But knowing the term *and* the ambient state *is* enough to determine the result; $\Downarrow : \mathcal{T} \times \mathcal{V} \rightharpoonup \mathcal{V}$

Some more examples:

$$\frac{\frac{\frac{}{\mathsf{ask} \mid 14 \Downarrow 14}}{\mathsf{local}\,14\,\mathsf{ask} \mid 1 \Downarrow 14} \quad \frac{}{\mathsf{ask} \mid 1 \Downarrow 1}}{\mathsf{local}\,14\,\mathsf{ask} + \mathsf{ask} \mid 1 \Downarrow 15}$$

$$\frac{\frac{\frac{}{\lambda a.a + \mathsf{ask} \mid 14 \Downarrow \lambda a.a + \mathsf{ask}}}{\mathsf{local}\,14\,(\lambda a.a + \mathsf{ask}) \mid 1 \Downarrow \lambda a.a + \mathsf{ask}} \quad \frac{}{\mathsf{ask} \mid 1 \Downarrow 1} \quad \frac{\frac{}{1 \mid 1 \Downarrow 1} \quad \frac{}{\mathsf{ask} \mid 1 \Downarrow 1}}{(a + \mathsf{ask})[1/a] \mid 1 \Downarrow 2}}{(\mathsf{local}\,14\,(\lambda a.a + \mathsf{ask}))\,\mathsf{ask} \mid 1 \Downarrow 2}$$

- Functions *delay* computation—the ask happens when the function body is evaluated, not the λ-term.

$$\frac{\frac{}{\lambda a.\mathsf{local}\,14\,(a + \mathsf{ask}) \mid 1 \Downarrow \lambda a.\mathsf{local}\,14\,(a + \mathsf{ask})} \quad \frac{}{\mathsf{ask} \mid 1 \Downarrow 1} \quad \frac{\frac{\frac{}{1 \mid 14 \Downarrow 1} \quad \frac{}{\mathsf{ask} \mid 14 \Downarrow 14}}{(1 + \mathsf{ask}) \mid 14 \Downarrow 15}}{(\mathsf{local}\,14\,(a + \mathsf{ask}))[1/a] \mid 1 \Downarrow 15}}{(\lambda a.\mathsf{local}\,14\,(a + \mathsf{ask}))\,\mathsf{ask} \mid 1 \Downarrow 15}$$

- Arguments are still evaluated at call sites.

But let's talk about call-by-name.

$$\frac{\frac{}{\lambda a.\mathsf{local}\,14\,(a + \mathsf{ask}) \mid 1 \Downarrow_{\mathsf{cbn}} \lambda a.\mathsf{local}\,14\,(a + \mathsf{ask})} \quad \frac{\frac{}{14 \mid 1 \Downarrow_{\mathsf{cbn}} 14} \quad \frac{\frac{}{\mathsf{ask} \mid 14 \Downarrow_{\mathsf{cbn}} 14} \quad \frac{}{\mathsf{ask} \mid 14 \Downarrow_{\mathsf{cbn}} 14}}{\mathsf{ask} + \mathsf{ask} \mid 14 \Downarrow_{\mathsf{cbn}} 28}}{(\mathsf{local}\,14\,(a + \mathsf{ask}))[\mathsf{ask}/a] \mid 1 \Downarrow_{\mathsf{cbn}} 28}}{(\lambda a.\mathsf{local}\,14\,(a + \mathsf{ask}))\,\mathsf{ask} \mid 1 \Downarrow_{\mathsf{cbn}} 28}$$

- In CBN, effects in *arguments* as well as functions may be delayed.