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:

\[
t ::= z \mid t \odot t \mid x \mid \lambda x.t \mid t t \mid \text{ask} \mid \text{local } t t
\]

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”:

\[
\frac{\text{ask} \mid r \downarrow r}{t \mid r' \downarrow v} \quad \frac{\text{local } r' t \mid r \downarrow v}{t \mid r' \downarrow v}
\]

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:

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

where (“predictably”):

\[
\text{ask}[v/x] = \text{ask} \quad (\text{local } r t)[v/x] = \text{local } (r[v/x]) t[v/x]
\]

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)
Some examples.

\[
\begin{array}{c|c|c}
\text{ask} & 1 & 1 \\
\hline
\text{ask + 1} & 1 & 2 \\
\end{array}
\quad
\begin{array}{c|c|c}
\text{ask} & 14 & 14 \\
\hline
\text{ask + 1} & 14 & 15 \\
\end{array}
\]

- 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; \(\downarrow: \mathcal{T} \times \mathcal{V} \to \mathcal{V}\)

Some more examples:

\[
\begin{array}{c|c|c}
\text{ask} & 14 & 14 \\
\hline
\text{local 14 ask} & 1 & 14 \\
\hline
\text{local 14 ask + ask} & 1 & 15 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
\lambda a. a + \text{ask} & 14 & 14 & \lambda a. a + \text{ask} & 1 & 14 \\
\hline
\text{local 14 (\lambda a. a + ask)} & 1 & 1 & \lambda a. a + \text{ask} & 1 & 1 \\
\hline
(\text{local 14 (\lambda a. a + ask)}) \text{ask} & 1 & 1 & 2 \\
\end{array}
\]

- Functions delay computation—the \text{ask} happens when the function body is evaluated, not the \(\lambda\)-term.

\[
\begin{array}{c|c|c|c|c|c}
\lambda a. \text{local 14 (a + ask)} & 1 & 1 & \lambda a. \text{local 14 (a + ask)} & 1 & 14 \\
\hline
\text{ask} & 1 & 1 & (\text{local 14 (a + ask)})[1/a] & 1 & 14 \\
\hline
(\lambda a. \text{local 14 (a + ask)}) \text{ask} & 1 & 1 & 2 \\
\end{array}
\]

- Arguments are still evaluated at call sites.

But let’s talk about call-by-name.

\[
\begin{array}{c|c|c|c|c|c}
\text{ask} & 14 & 14 & 14 & \text{ask} & 14 & 14 \\
\hline
\text{local 14 (a + ask)} & 1 & \text{local 14 (a + ask)} & 1 & \text{local 14 (a + ask)} & 1 & \text{local 14 (a + ask)} \\
\hline
(\lambda a. \text{local 14 (a + ask)}) \text{ask} & 1 & 1 & 15 & 28 \\
\end{array}
\]

- In CBN, effects in \textit{arguments} as well as functions may be delayed.