Day 14.

1. Type and Effect Systems

Let’s consider a language with state and exceptions. (Exact semantics unimportant—we’ll do with an intuitive semantics for now.)

\[ e ::= z \mid e \odot e \mid x \mid \lambda x.e \mid e e \mid \text{get} \mid \text{put} \mid \text{throw} \mid \text{try e catch e} \]

Now, we want to design a type system—that is, a static approximation of its (intuitive) dynamic semantics—for this language.

- Initial intuition: what does \( \Gamma \vdash t : T_1 \rightarrow T_2 \) mean? It means that \( t \) defines a term that, given a \( T_1 \) shaped argument, produces a \( T_2 \) shaped result. That is, it approximates the observable behavior of \( t \).
- Just types sufficient for pure functional programming: intuitively, nothing but the free variables of a term (i.e., \( \Gamma \)) determine the meaning of the term.
- Insufficient for impure functional programming... but why would we care?
  - Correctness: can we do two things in parallel?
  - Compiler transformations: can we combine subexpressions? Omit dead code? Etc.

Goal: a type and effect system which characterizes both what a term produces and how it produces it.

\[ \mathcal{T} \ni t ::= \text{Int} \mid t \rightarrow t \]
\[ \mathcal{F} \ni f ::= \text{get} \mid \text{put} \mid \text{throw} \]

Our typing relation will now be a 4-place relation, associating a term and its context with both a type \( (t \in \mathcal{T}) \) and a set of effects \( (F \subseteq \mathcal{F}) \).

\[ \cdot \vdash \cdot : \cdot \& \cdot \subseteq (\mathcal{X} \rightarrow \mathcal{T}) \times \mathcal{E} \times \mathcal{T} \times \mathcal{P}(\mathcal{F}) \]

Let’s try to write some typing rules.

We’ll start with integers.

\[
\begin{align*}
\Gamma \vdash e_1 : \text{Int} & \quad \Gamma \vdash e_2 : \text{Int} & \quad \Gamma \vdash e_3 : \text{Int} & \quad \Gamma \vdash e_4 : \text{Int} & \quad \Gamma \vdash e_5 : \text{Int} \\
\Gamma \vdash z : \text{Int} & \\
\Gamma \vdash e_1 \odot e_2 : \text{Int} & \\
\Gamma \vdash e_3 \odot e_4 : \text{Int} & \\
\Gamma \vdash e_5 \odot e_6 : \text{Int} & \\
\Gamma \vdash e_7 \odot e_8 : \text{Int} & \\
\Gamma \vdash e_9 \odot e_{10} : \text{Int} & \\
\Gamma \vdash e_{11} \odot e_{12} : \text{Int} &
\end{align*}
\]

- Integer constants “obviously” have no effect.
- Binary operations have as many effects as their operands do... for example, \( \text{get} + 1 \) must have the effects that \( \text{get} \) does, but it doesn’t add any more effects of its own.

Now let’s look at some side-effecting operations:

\[
\begin{align*}
\Gamma \vdash \text{get} : \text{Int} & \quad \Gamma \vdash \text{put} : \text{Int} \\
\Gamma \vdash \text{get} & \quad \Gamma \vdash \text{put} \quad \Gamma \vdash : \text{Int} & \quad \Gamma \vdash : \text{Int} &
\end{align*}
\]
We’re making a simplifying assumption here: that the state is always an integer. We’ll do the same for throw/catch. This isn’t necessary, but it is a significant simplification at this point—otherwise, we would have to track changes in the type of the state through a program.

- **get** has a side effect—it reads the state—so we reflect that in its effects.
- **put** has a side effect—it writes the state—so we reflect that in its effects. But it also has any side effect that its argument term \( t \) would have. For example, \( \text{put}(\text{get} + 1) \) both reads and writes the state.
- **get** and **put** effects accumulate, but never go away. (We don’t have any idea of a “local” state invisible to the outside world. But we could do... what might that look like?)

How about exceptions?

\[
\begin{align*}
\Gamma \vdash e : \text{Int} & \quad F \\
\Gamma \vdash \text{throw } e : t & \quad F \cup \{\text{throw}\} \\
\Gamma \vdash e_1 : t & \quad F_1 \\
\Gamma \vdash e_2 : \text{Int} & \quad t \quad F_2 \\
\Gamma \vdash \text{try } e_1 \text{ catch } e_2 : t & \quad (F_1 \setminus \{\text{throw}\}) \cup F_2
\end{align*}
\]

- Again, we assume that the thrown value is an Int; this simplifies the typing of try . . . catch . . . (Although it is much easier here to imagine how to adapt the effect system to thrown values of any type. How would you do it?)
- **throw** \( e \) has any effects that \( e \) has: **throw** get, for example, both reads the state and thrown an exception.
- **throw** \( e \) has an arbitrary return type. Why is this justified? Why is this necessary?
- In **try** \( e_1 \text{ catch } e_2 \), we don’t know whether \( e_2 \) will execute, so we include its effects regardless. But, we can filter **throw** from the effects of \( e_1 \), since if \( e_1 \) did throw then it would be caught. (This does not mean that the effects of **try** \( e_1 \text{ catch } e_2 \) may not include **throw**. Why?)