Day 15.

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.)

\[
t ::= z \mid t \mathbin{\circ} t \mid x \mid \lambda x.t \mid t t \mid \text{get} \mid \text{put} t \mid \text{throw} t \mid \text{try} t \text{catch} t
\]

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 \( \Gamma \vdash t : T_1 \to T_2 \) 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.

\[
\mathcal{Y} \ni T ::= \text{Int} \mid T \to T
\]

\[
\mathcal{E} \ni e ::= \text{get} \mid \text{put} \mid \text{throw}
\]

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

\[
\vdash \cdot \cdot : \cdot \cdot \subseteq (\mathcal{X} \to \mathcal{Y}) \times \mathcal{T} \times \mathcal{Y} \times \mathcal{P}(\mathcal{E})
\]

Let’s try to write some typing rules.

We’ll start with integers.

\[
\begin{align*}
\Gamma \vdash z : \text{Int} & \quad \Gamma \vdash t : \text{Int} & \quad \Gamma \vdash e : \text{Int} & \\
\Gamma \vdash t_1 \cdot t_2 : \text{Int} & \quad \Gamma \vdash e_1 & \quad \Gamma \vdash e_2
\end{align*}
\]

- 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:

\[
\begin{align*}
\Gamma \vdash \text{get} : \text{Int} & \quad \Gamma \vdash t : \text{Int} & \quad \Gamma \vdash t : \text{Int} & \quad \Gamma \vdash \text{put} t : \text{Int} & \quad \Gamma \vdash \text{put} t : \text{Int}
\end{align*}
\]
- 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, `put(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?

\[ \Gamma \vdash t : \text{Int} \& e \quad \Gamma \vdash t_1 : T \& e_1 \quad \Gamma \vdash t_2 : \text{Int} \to T \& e_2 \]
\[ \Gamma \vdash \text{try } t_1 \text{ catch } t_2 : T \& (e_1 \setminus \{ \text{throw} \}) \cup e_2 \]

- 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 t` has any effects that `t` has: `throw get`, for example, both reads the state and thrown an exception.
- `throw t` has an arbitrary return type. Why is this justified? Why is this necessary?
- In `try t_1 catch t_2`, we don’t know whether `t_2` will execute, so we include its effects regardless. But, we can filter `throw` from the effects of `t_1`, since if `t_1` did throw then it would be caught. (This does not mean that the effects of `try t_1 catch t_2` may not include `throw`. Why?)

Now we can do the “obvious” thing for functions.

\[ \Gamma \vdash x : \Gamma(x) \& \emptyset \quad \Gamma \vdash \lambda x. t : T_1 \to T_2 \& \emptyset \quad \Gamma \vdash t_1 : T_1 \to T_2 \& e_1 \quad \Gamma \vdash t_2 : T_1 \& e_2 \]

- No effects in defining a function—recall the local example.
- Application combines left and right effects—\((\lambda a.\lambda b.a + b)(\text{put } 1)\)\text{get} has both get (rhs) and put (lhs) effects.

Let’s see it work:

\[ \{a \mapsto \text{Int}\} \vdash a : \text{Int} \& \emptyset \quad \{a \mapsto \text{Int}\} \vdash 1 : \text{Int} \& \emptyset \]
\[ \{a \mapsto \text{Int}\} \vdash a + 1 : \text{Int} \& \emptyset \]
\[ \{a \mapsto \text{Int}\} \vdash \text{put}(a + 1) : \text{Int} \& \{ \text{put} \} \]
\[ \emptyset \vdash \lambda a.\text{put}(a + 1) : \text{Int} \to \text{Int} \& \emptyset \]
\[ \emptyset \vdash (\lambda a.\text{put}(a + 1))1 : \text{Int} \& \emptyset \]

Something seems to have gone wrong: intuitively, we should expect that evaluating this term will have a `put` effect. But that’s vanished from its type.

Key idea: we’ve lost track of the effects that happen when executing the body of the function—the `e` above the line in the λ typing rule appears nowhere below the line. That effect shouldn’t happen when we define the function, but we need to keep track of it for each use of the function.

\[ \mathcal{Y} \ni T ::= \text{Int} \mid T \overline{E} T \quad (E \subseteq \mathcal{E}) \]
Now we can restate the typing rules for functions:

\[
\begin{align*}
\Gamma[x \mapsto T_1] & \vdash t : T_2 \land e \\
\Gamma & \vdash \lambda x. t : T_1 \rightarrow T_2 \land \emptyset \\
\Gamma & \vdash t_1 : T_1 \rightarrow T_2 \land e_1 \quad \Gamma & \vdash t_2 : T_1 \land e_2 \\
\Gamma & \vdash t_1 \; t_2 : T_2 \land e_1 \cup e_2 \cup e_3
\end{align*}
\]

And our example should work:

\[
\begin{align*}
\{a \mapsto \text{Int}\} & \vdash a : \text{Int} \land \emptyset \\
\{a \mapsto \text{Int}\} & \vdash 1 : \text{Int} \land \emptyset \\
\{a \mapsto \text{Int}\} & \vdash a + 1 : \text{Int} \land \emptyset \\
\{a \mapsto \text{Int}\} & \vdash \text{put}(a + 1) : \text{Int} \land \{\text{put}\} \\
\emptyset & \vdash \lambda a. \text{put}(a + 1) : \text{Int} \land \{\text{put}\} \\
\emptyset & \vdash 1 : \text{Int} \land \emptyset \\
\emptyset & \vdash (\lambda a. \text{put}(a + 1)) \; 1 : \text{Int} \land \{\text{put}\}
\end{align*}
\]