Recursive Types and General Recursion

Stefan Muller

CSE 5095: Types and Programming Languages, Fall 2025
Lecture 14

1 General Recursion

As we saw a couple weeks ago, STLC programs all terminate, which makes it not a very useful language. The reason, as we saw, is that we can't write terms involving self-application, like $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$ and fixed point combinators like Y. But we can explicitly add a fixed point combinator to the language.

$$\textit{Expressions} \quad e \quad ::= \quad \cdots \mid \mathsf{fix}\ x = e$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\ x = e : \tau}\ (\textsc{Fix}) \qquad \frac{}{\mathsf{fix}\ x = e \mapsto [\mathsf{fix}\ x = e/x]e}\ (\textsc{StepFix})$$

Now we can write infinite loops again!

$$\mathsf{fix}\ x = x \mapsto \mathsf{fix}\ x = x \mapsto \ldots$$

We can also write useful recursive programs.

$$\textit{fact} \triangleq \mathsf{fix}\ \textit{fact} = \lambda n : \mathsf{int.case}\ n \leq \overline{1}\ \mathsf{of}\ \{x.\overline{1}; y.n * (\textit{fact}\ (n - \overline{1}))\}$$

$$\begin{array}{ll} & (\mathsf{fix}\ \textit{fact} = \lambda n : \mathsf{int.case}\ n \leq \overline{1}\ \mathsf{of}\ \{x.\overline{1}; y.n * (\textit{fact}\ (n - \overline{1}))\})\ \overline{2} \\ \mapsto & (\lambda n : \mathsf{int.case}\ n \leq \overline{1}\ \mathsf{of}\ \{x.\overline{1}; y.n * (\textit{fact}\ (n - \overline{1}))\})\ \overline{2} \\ \mapsto & \mathsf{case}\ \overline{2} \leq \overline{1}\ \mathsf{of}\ \{x.\overline{1}; y.n * (\textit{fact}\ (n - \overline{1}))\} \\ \mapsto & \mathsf{case}\ \mathsf{inr}\ ()\ \mathsf{of}\ \{x.\overline{1}; y.n * (\textit{fact}\ (n - \overline{1}))\} \\ \mapsto & \overline{2} * (\textit{fact}\ (\overline{2} - \overline{1})) \\ \mapsto & \overline{2} * (\textit{fact}\ \overline{1}) \\ \mapsto & \overline{2} * ((\lambda n : \mathsf{int.case}\ n \leq \overline{1}\ \mathsf{of}\ \{x.\overline{1}; y.n * (\textit{fact}\ (n - \overline{1}))\})\ \overline{1}) \\ \mapsto & \overline{2} * (\mathsf{case}\ \overline{1} \leq \overline{1}\ \mathsf{of}\ \{x.\overline{1}; y.n * (\textit{fact}\ (n - \overline{1}))\}) \\ \mapsto & \overline{2} * (\mathsf{case}\ \mathsf{inl}\ ()\ \mathsf{of}\ \{x.\overline{1}; y.n * (\textit{fact}\ (n - \overline{1}))\}) \\ \mapsto & \overline{2} * \overline{1} \\ \mapsto & \overline{2} \end{array}$$

2 Recursive Types

If we really want a good model of functional languages, we need more interesting types, in particular to be able to express recursive data structures like linked lists. Let's try to build a type $\mathsf{intlist}$ which has two introduction forms: nil is the empty list, and $\mathsf{cons}\ (\overline{n}, l)$ adds $\overline{n}$ to the front of l, where l is another $\mathsf{intlist}$. The fact that there are two ways to build it suggests using sum types:

$$\begin{array}{rcl} \mathsf{intlist} & \triangleq & \mathsf{unit} + (\mathsf{int} \times \mathsf{intlist}) \\ \mathsf{nil} & \triangleq & \mathsf{inl}\ () \\ \mathsf{cons}\ (e_1, e_2) & \triangleq & \mathsf{inr}\ (e_1, e_2) \end{array}$$

But this definition is circular, so it doesn't work. We need some way of defining recursive types.

$$\tau \quad ::= \quad \cdots \mid \alpha \mid \mu\alpha.\tau$$

In $\mu\alpha.\tau$, we define the type we want as τ, in which α is bound to a recursive instance of the type. We call α a type variable: it stands for a type, rather than an expression (and we'll substitute types for it). So, for example:

$$\mathsf{intlist} \triangleq \mu\alpha.\mathsf{unit} + (\mathsf{int} \times \alpha)$$

As an aside, this means we now have a way to make types with unbound type variables, like $\alpha \times \mathsf{unit}$. We need a judgment $\Delta \vdash \tau\ \mathsf{ok}$ that says that τ doesn't have unbound type variables. Here, Δ is a context that has the type variables we're allowed to use.

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha\ \mathsf{ok}} \qquad \frac{\Delta, \alpha \vdash \tau\ \mathsf{ok}}{\Delta \vdash \mu\alpha.\tau\ \mathsf{ok}} \qquad \frac{}{\Gamma \vdash \mathsf{unit}\ \mathsf{ok}} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{ok} \qquad \Delta \vdash \tau_2\ \mathsf{ok}}{\Delta \vdash \tau_1 \to \tau_2\ \mathsf{ok}} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{ok} \qquad \Delta \vdash \tau_2\ \mathsf{ok}}{\Delta \vdash \tau_1 \times \tau_2\ \mathsf{ok}}$$

$$\frac{\Delta \vdash \tau_1\ \mathsf{ok} \qquad \Delta \vdash \tau_2\ \mathsf{ok}}{\Delta \vdash \tau_1 + \tau_2\ \mathsf{ok}}$$

We'll talk more about this a bit later.

So, does that mean we just have, e.g., $\mathsf{nil} \triangleq \mathsf{inl}\ ()$? Not quite. $\mathsf{inl}\ ()$ has type $\mathsf{unit} + (\mathsf{int} \times \mathsf{intlist})$, not $\mu\alpha.\mathsf{unit} + (\mathsf{int} \times \alpha)$, which is how we defined $\mathsf{intlist}$. So we need a way of converting between these two types.

First, note that we want to perform the following substitution:

$$[\mathsf{intlist}/\alpha](\mathsf{unit} + (\mathsf{int} \times \alpha))$$

or, expanding:

$$[(\mu\alpha.\mathsf{unit} + (\mathsf{int} \times \alpha))/\alpha](\mathsf{unit} + (\mathsf{int} \times \alpha))$$

$$e \quad ::= \quad \cdots \mid \mathsf{roll}_\tau\ e \mid \mathsf{unroll}\ e$$

$$\frac{\Gamma \vdash e : [\mu\alpha.\tau/\alpha]\tau}{\Gamma \vdash \mathsf{roll}_{\mu\alpha.\tau}\ e : \mu\alpha.\tau}\ (\mu\text{-I}) \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathsf{unroll}\ e : [\mu\alpha.\tau/\alpha]\tau}\ (\mu\text{-E})$$

The roll and unroll expressions are really just there to make the typing rules work out, but they're in the expressions, so we need step rules for them:

$$\frac{e\ \mathsf{val}}{\mathsf{roll}_\tau\ e\ \mathsf{val}}\ (\textsc{ValRoll}) \qquad \frac{e \mapsto e'}{\mathsf{roll}_\tau\ e \mapsto \mathsf{roll}_\tau\ e'}\ (\textsc{StepSearchRoll})$$

$$\frac{e \mapsto e'}{\mathsf{unroll}\ e \mapsto \mathsf{unroll}\ e'}\ (\textsc{StepSearchUnroll}) \qquad \frac{e\ \mathsf{val}}{\mathsf{unroll}\ \mathsf{roll}_\tau\ e \mapsto e}\ (\textsc{StepUnroll})$$

Basically, unroll just cancels out a roll. Here are the correct definitions for int lists:

$$\begin{aligned} \mathsf{intlist} &\triangleq \mu\alpha.\mathsf{unit} + (\mathsf{int} \times \alpha) \\ \mathsf{nil} &\triangleq \mathsf{roll}_{\mathsf{intlist}}\ (\mathsf{inl}\ ()) \\ \mathsf{cons}\ (e_1, e_2) &\triangleq \mathsf{roll}_{\mathsf{intlist}}\ (\mathsf{inr}\ (e_1, e_2)) \end{aligned}$$

We use unroll if we want to use integer lists. For example, the following function returns the first element of a list or 0 if it's empty:

$$\lambda x : \mathsf{intlist.case\ unroll}\ x\ \mathsf{of}\ \{\_.\overline{0}; p.\mathsf{fst}\ p\}$$

We can combine this with the fixed point operator to make a function that adds 1 to every element of a list:

$$\mathsf{fix}\ f = \lambda x : \mathsf{intlist.case\ unroll}\ x\ \mathsf{of}\ \{\_.\mathsf{roll}_{\mathsf{intlist}}\ \mathsf{inl}\ ();p.\mathsf{roll}_{\mathsf{intlist}}\ \mathsf{inr}\ (\mathsf{fst}\ p + \overline{1}, f\ (\mathsf{snd}\ p))\}$$

Note that, to use something of a recursive type, we have to unroll it to turn it into the actual underlying type. Just before we return something of a recursive type, we roll it so that we have the recursive type backs. This pattern works most of the time.

As it turns out, we didn't actually need fix: just STLC with recursive types is enough to not only encode general recursion, but actually all of the untyped lambda calculus. This is because we can define the "type" of "untyped" lambda terms:

$$\mathsf{D} \triangleq \mu\alpha.\alpha \to \alpha$$

We have:

$$\begin{array}{lcl} \lambda x : \mathsf{D}.(\mathsf{unroll}\ x)\ x & : & \mathsf{D} \to \mathsf{D} \\ \mathsf{roll}_{\mathsf{D}}\ (\lambda x : \mathsf{D}.(\mathsf{unroll}\ x)\ x) & : & \mathsf{D} \end{array}$$

So we can encode our infinite loop lambda term $(\lambda x.x\ x)(\lambda x.x\ x)$ as

$$(\lambda x : \mathsf{D}.(\mathsf{unroll}\ x)\ x)\ (\mathsf{roll}_{\mathsf{D}}\ (\lambda x : \mathsf{D}.(\mathsf{unroll}\ x)\ x))$$