

Recursive Types and General Recursion

Stefan Muller

CS 534: Types and Programming Languages, Spring 2024
Lecture 15

1 General Recursion

As we saw above, STLC programs all terminate, which makes it not a very useful language. We need some sort of fixed-point combinator again, but we obviously can't write one in STLC. To see why, remember our self-application trick. Let's try to figure out the type of $(\lambda x.x x)$ $(\lambda x.x x)$. There's no type annotation, so let's just say the type of x is τ and we'll figure it out later. We'll also say the return type of each function is τ' .

$$\frac{\frac{\frac{}{x : \tau \vdash x : \tau \rightarrow \tau} \text{ (???)} \quad x : \tau \vdash x : \tau}{x : \tau \vdash x x : \tau'}}{\bullet \vdash \lambda x.x x : \tau \rightarrow \tau'} \quad \dots}{\bullet \vdash (\lambda x.x x) (\lambda x.x x) : ???}$$

But we can add it to the language.

Expressions $e ::= \dots \mid \text{fix } x = e \text{ xif}$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{fix } x = e \text{ xif} : \tau} \text{ (Fix)} \qquad \frac{}{\text{fix } x = e \text{ xif} \mapsto [\text{fix } x = e \text{ xif}/x]e} \text{ (S-Fix)}$$

Now we can write infinite loops again!

$$\text{fix } x = x \text{ xif} \mapsto \text{fix } x = x \text{ xif} \mapsto \dots$$

We can also write useful recursive programs.

$$\begin{aligned} \text{fact} &\triangleq \text{fix fact} = \lambda n : \text{int}. \text{case } n \leq \bar{1} \text{ of } \{x.\bar{1}; y.n * (\text{fact } (n - \bar{1}))\} \text{ xif} \\ &\mapsto (\text{fix fact} = \lambda n : \text{int}. \text{case } n \leq \bar{1} \text{ of } \{x.\bar{1}; y.n * (\text{fact } (n - \bar{1}))\} \text{ xif}) \bar{2} \\ &\mapsto (\lambda n : \text{int}. \text{case } n \leq \bar{1} \text{ of } \{x.\bar{1}; y.n * (\text{fact } (n - \bar{1}))\}) \bar{2} \\ &\mapsto \text{case } \bar{2} \leq \bar{1} \text{ of } \{x.\bar{1}; y.n * (\text{fact } (n - \bar{1}))\} \\ &\mapsto \text{case inr } () \text{ of } \{x.\bar{1}; y.n * (\text{fact } (n - \bar{1}))\} \\ &\mapsto \bar{2} * (\text{fact } (\bar{2} - \bar{1})) \\ &\mapsto \bar{2} * (\text{fact } \bar{1}) \\ &\mapsto \bar{2} * ((\lambda n : \text{int}. \text{case } n \leq \bar{1} \text{ of } \{x.\bar{1}; y.n * (\text{fact } (n - \bar{1}))\}) \bar{1}) \\ &\mapsto \bar{2} * (\text{case } \bar{1} \leq \bar{1} \text{ of } \{x.\bar{1}; y.n * (\text{fact } (n - \bar{1}))\}) \\ &\mapsto \bar{2} * (\text{case inl } () \text{ of } \{x.\bar{1}; y.n * (\text{fact } (n - \bar{1}))\}) \\ &\mapsto \bar{2} * \bar{1} \\ &\mapsto \bar{2} \end{aligned}$$

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 `intlist` which has two introduction forms: `nil` is the empty list, and `cons` (\bar{n}, l) adds \bar{n} to the front of l , where l is another `intlist`. The fact that there are two ways to build it suggests using sum types:

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

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

$$\tau ::= \dots \mid \alpha \mid \mu\alpha.\tau$$

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:

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

As an aside, this means we now have a way to make types with unbound type variables, like $\alpha \times \text{unit}$. We need a judgment $\Delta \vdash \tau \text{ ok}$ 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.

$$\begin{array}{c} \frac{\alpha \in \Delta}{\Delta \vdash \alpha \text{ ok}} \quad \frac{\Delta, \alpha \vdash \tau \text{ ok}}{\Delta \vdash \mu\alpha.\tau \text{ ok}} \quad \frac{}{\Gamma \vdash \text{unit ok}} \quad \frac{\Delta \vdash \tau_1 \text{ ok} \quad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ ok}} \quad \frac{\Delta \vdash \tau_1 \text{ ok} \quad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 \times \tau_2 \text{ ok}} \\ \\ \frac{\Delta \vdash \tau_1 \text{ ok} \quad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 + \tau_2 \text{ ok}} \end{array}$$

We'll talk more about this a bit later.

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

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

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

or, expanding:

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

$$e ::= \dots \mid \text{roll}_\tau e \mid \text{unroll } e$$

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

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:

$$\frac{e \text{ val}}{\text{roll}_\tau e \text{ val}} \quad \frac{e \mapsto e'}{\text{roll}_\tau e \mapsto \text{roll}_\tau e'} \quad \frac{e \mapsto e'}{\text{unroll } e \mapsto \text{unroll } e'} \quad \frac{e \text{ val}}{\text{unroll roll}_\tau e \mapsto e}$$

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

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

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:

$$\lambda x : \text{intlist.case } x \text{ of } \{ _.\bar{0}; p.\text{fst } p \}$$

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

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

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:

$$D \triangleq \mu\alpha.\alpha \rightarrow \alpha$$

We have:

$$\begin{aligned} \lambda x : D.(\text{unroll } x) x & : D \rightarrow D \\ \text{roll}_D (\lambda x : D.(\text{unroll } x) x) & : D \end{aligned}$$

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

$$(\lambda x : D.(\text{unroll } x) x) (\text{roll}_D (\lambda x : D.(\text{unroll } x) x))$$