Paul Downen [2018/07/05]

(Lect 3) **Recursion & Recursive Types**

Abstr. Syntax of Expressions (for numbers)

$$e ::= \ldots \mid z \mid s(e) \mid rec(e\,;\, \underbrace{e_0}_{\text{what to do in 0 case}}\,;\, \overset{\text{(point at which IH is given)}}{x},\, \underbrace{y.e_1}_{\text{given result at } x\text{, how to get result at } x+1})$$

(slightly) more intuitive syntax:

$$rec\ e\ as\ \{ z \Rightarrow e_0 \mid S\,x\ with\ y \Rightarrow e_1 \}$$

- Example 1

$$add\ z\ n = n$$

$$add\ S(m)\ n = S(add\ m\ n)$$

Q: How to encode this if we just have recursors & lambdas?

A: $add = \lambda m{:}Nat.\ \lambda n{:}Nat.\ rec\ m\ as\ \{z \Rightarrow n \mid Sm'\ with\ r \Rightarrow Sr\}$

- Example 2

$$mult\ z\ n = z$$

$$mult\ (Sm)\ n = add\ n\ (mult\ m\ n)$$

Encoding:

$$mult = \lambda m{:}Nat.\ \lambda n{:}Nat.\ rec\ m\ as$$

$$z \Rightarrow z$$

$$Sm'\ with\ r \Rightarrow add\ n\ r$$

- Example 3

$$\mathrm{pred}\ z = z$$
$$\mathrm{pred}\ (S\ n) = n$$

Encoding:

$$\mathrm{pred} = \lambda n : Nat.\ \mathrm{rec}\ n\ \mathrm{as}\ \begin{array}{l} z \Rightarrow z \\ Sn' \ \mathrm{with}\ r \Rightarrow n' \end{array}$$

Types & Typing Rules

$$\tau ::= \ldots \mid Nat$$

$$\frac{}{\Gamma \vdash z : Nat} \qquad \frac{\Gamma \vdash e : Nat}{\Gamma \vdash S(e) : Nat}$$

$$\frac{\Gamma \vdash e : Nat \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : Nat, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathrm{rec}\ e\ \mathrm{as}\ \{ z \Rightarrow e_0 \mid Sx\ \mathrm{with}\ y \Rightarrow e_1 \} : \tau}$$

Dynamic Semantics (lazy version)

could do call by val or eager, but it's a bit different

$$\frac{}{z\ val}$$

$$\frac{}{Se\ val}$$

or

$$\frac{e\ val}{Se\ val}$$

↑ eager semantics

$$\frac{}{\mathrm{rec}\ z\ \mathrm{as}\ \begin{array}{l} z \Rightarrow e_0 \\ Sx\ \mathrm{with}\ y \Rightarrow e_1 \end{array} \longmapsto e_0}$$

$$\frac{}{\mathrm{rec}\ Se\ \mathrm{as}\ \begin{array}{l} z \Rightarrow e_0 \\ Sx\ \mathrm{with}\ y \Rightarrow e_1 \end{array} \longmapsto e_1 \left[e/x \,,\ \underbrace{\left(\mathrm{rec}\ e\ \mathrm{as}\ \begin{array}{l} z \Rightarrow e_0 \\ Sx\ \mathrm{with}\ y \Rightarrow e_1 \end{array} \right)}_{\text{this would also change in eager semantics}} / y \right]}$$

Evaluation context:

$$E ::= \dots \mid \text{rec } \underline{E} \text{ as } \{\dots\} \mid \underbrace{S\,E}_{\text{eager semantics}}$$

$$\frac{e \longmapsto e'}{\begin{array}{l}\text{rec } e \text{ as}\\ \quad z \Rightarrow e_0\\ \quad Sx \text{ with } y \Rightarrow e_1\end{array} \longmapsto \begin{array}{l}\text{rec } e' \text{ as}\\ \quad z \Rightarrow e_0\\ \quad Sx \text{ with } y \Rightarrow e_1\end{array}}$$

Recursive Types More Generally

Recursive type variable (Type recursion, not polymorphism)

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

$$\mu\alpha.\tau = \tau[{}^{\mu\alpha.\tau}/_{\alpha}] \quad \text{(equirecursion)}$$ harder to deal with

$$\mu\alpha.\tau \underset{\text{unfold}}{\overset{\text{fold}}{\cong}} \tau[{}^{\mu\alpha.\tau}/_{\alpha}] \quad \text{(isorecursion)}$$ easier

They're not the same in isorecursion, but each can be transformed into the other.

$$e ::= \dots \mid \text{fold}(e) \mid \text{unfold}(e)$$

$$\frac{\Gamma \vdash e : \tau[{}^{\mu\alpha.\tau}/_{\alpha}]}{\Gamma \vdash \text{fold}(e) : \mu\alpha.\tau} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \text{unfold}(e) : \tau[{}^{\mu\alpha.\tau}/_{\alpha}]}$$

(Intro) (Elim)

Rules for running fold/unfold

$\boxed{\text{CBN}}$

$$\frac{}{\text{unfold}(\text{fold}(e)) \longmapsto e} \qquad \frac{}{\text{fold } e \text{ val}}$$

$$\frac{e \longmapsto e'}{\text{unfold}(e) \longmapsto \text{unfold}(e')}$$

$$(\eta) \quad \text{fold}(\text{unfold}(e)) =_\eta e : \mu\alpha.\tau$$

$\boxed{\text{CBV}}$

$$\frac{e \text{ val}}{\text{fold}(e) \text{ val}}$$

$$\frac{e \text{ val}}{\text{unfold}(\text{fold}(e)) \longmapsto e}$$

$$\frac{e \longmapsto e'}{\text{fold}(e) \longmapsto \text{fold}(e')}$$ ... + other one.

Example 1

$\text{Nat} \approx \text{unit} + \text{Nat}$ But this is "circular"

Instead, $\text{Nat} = \mu\alpha.\,\text{unit} + \alpha$

$z = \text{fold}(l \cdot \langle\rangle)$

$S\,e = \text{fold}(r \cdot e)$

Example 2

$\text{List } \tau \approx \text{unit} + (\tau \times \text{List } \tau)$

$\text{List } \tau = \mu\alpha.(\text{unit} + (\tau \times \alpha))$

$\text{nil} = \text{fold}(l \cdot \langle\rangle)$

$\text{cons } e\ e' = \text{fold}(r \cdot \langle e, e' \rangle)$

But now that we've introduced recursion we have introduced a type that can be applied to itself.

let $\omega : \mu\alpha(\alpha \to \tau) \to \tau$

$$\omega = \lambda x.\, \mu\alpha.(\alpha \to \tau) . \underbrace{(\mathrm{unfold}\ x)}_{(\mu\alpha.\alpha \to \tau) \to \tau}\ x$$

$\Omega : \tau$

$$\Omega = \omega\,(\underbrace{\mathrm{fold}}_{\mu\alpha.\alpha \to \tau}\ \omega)$$

Then

$$\begin{tikzcd}
\omega \arrow[r, phantom, "\approx"] & \mu\alpha.(\alpha \to \tau) \to \tau \arrow[r, phantom, "\approx"] \arrow[r, bend right, "\mathrm{fold}"'] & \mu\alpha.(\alpha \to \tau) \arrow[l, bend right, "\mathrm{unfold}"']
\end{tikzcd}$$

So we've introduced recursive programs by adding recursive types.

The "right" way to express recursion is with the Y-combinator, like this:

$$Y : (\tau \to \tau) \to \tau$$

$$Y = \lambda f : \tau \to \tau .\, (\lambda x : \mu\alpha.(\alpha \to \tau).\, f(\mathrm{unfold}\ x\ x))$$
$$(\mathrm{fold}\,(\lambda x : \mu\alpha.(\alpha \to \tau).\, f(\mathrm{unfold}\ x)))$$