Paul Downen (Lect 2)

[2018/06/03]

The λ-calculus (Church - 1930's)

- Syntax of expressions of the λ-calculus

$$e ::= x \mid e_1 e_2 \mid \lambda x.e$$

Or we could specify the syntax using trees:

$$e ::= x \mid app(e_1, e_2) \mid lam(x.e)$$

- Semantics (λ-calculus "laws")

(α) $\lambda x.e =_\alpha \lambda y.([y/x]e)$ $(y \notin FV(e))$

(β) $(\lambda x.e)\, e' =_\beta [e'/x]e$

(η) $\lambda x.(e\, x) =_\eta e$ $(x \notin FV(e))$

- Dynamic Semantics of the λ-calculus

(call by name)

$$(\lambda x.e)\, e' \longmapsto e[e'/x]$$

what about expressions like $((\lambda x. \lambda y. x)\, 1)\, 2$?

$$\frac{e_1 \longmapsto e_1'}{e_1 e_2 \longmapsto e_1' e_2}$$

so

$$\frac{(\lambda x. \lambda y. x)\, 1 \longmapsto \lambda y. 1}{((\lambda x. \lambda y. x)\, 1)\, 2 \longmapsto (\lambda y. 1)\, 2}$$

Paul Downen

(Lect 2) [2018/06/03]

Evaluation Contexts

$$E \in EvalCtx ::= \Box \mid E\,e$$

$$\frac{e \longmapsto e'}{E[e] \longmapsto E[e']} \qquad \Box[e] = e$$

$$(E\,e')[e] = (E[e])\,e'$$

Call-by-Value

$(\lambda x.e)\,V \longmapsto e[V/x]$ where V is a value

Q: What is a value? A: Everything that's not an application.

$V \in Value ::= x \mid \lambda x.e$ (But sometimes it's defined by $Value ::= \lambda x.e$)

How do we find all the "redexes" (reducible components)?

$$E \in EvalCtx ::= \Box \mid E\,e \mid V\,E$$

Inference rule:

$$\frac{e \longmapsto e'}{E[e] \longmapsto E[e']}$$

(If the 1st component is already a value V, work on the 2nd component.)

Reduction Rules

$$\frac{}{x\ val} \qquad \frac{}{\lambda x.e\ val} \qquad \frac{e'\ val}{(\lambda x.e)\,e' \longmapsto e[e'/x]}$$

$$\frac{e_1 \longmapsto e_1'}{e_1 e_2 \longmapsto e_1' e_2} \qquad \frac{e_1\ val \quad e_2 \longmapsto e_2'}{e_1 e_2 \longmapsto e_1 e_2'}$$

These rules define a call-by-name semantics that's equivalent to that given by ~~$\not\beta$~~ above.

$\boxed{\text{If/then/else}}$ (How to encode ite, tt, ff in λ-cal)

$$\text{if } e \text{ then } e_1 \text{ else } e_2 = (e\ e_1)\ e_2$$

$$\text{True} := \lambda x\, \lambda y.\, x$$

$$\text{False} := \lambda x\, \lambda y.\, y$$

(The "main" op on Booleans is if/then/else.)

Homework Encode not in λ-calc.

EX **Encoding Sets**

Define $e \in e'$ by $e' e$

$$e_1 \cup e_2 := \lambda x.\, \text{or}\, (e_1 x)(e_2 x)$$

$$e_1 \cap e_2 = \lambda x.\, \text{and}\, (e_1 x)(e_2 x)$$

Russel's Paradox

Let $R = \{ e : Set \mid e \notin e \}$

Then $R \in R$ has no ans T/F.

In λ-calc, $R = \lambda x. \, not(x x)$

Then $RR \longmapsto (not(x x))[R/x] = not \, RR$

$\longmapsto not \; not \; RR$

$\longmapsto \dots \longmapsto not \dots not \; RR$

$\longmapsto \dots$

So λ and app adds a new "feature" to our language that we didn't intend. i.e. looping forever.

Let $\Omega = (\lambda x. x x)(\lambda x. x x)$

Then $\Omega \longmapsto \Omega$.

So what if, instead of negation of self-app, like we had in Russel's, we introduce

$$Y f = (\lambda x. f(x x))(\lambda x. f(x x))$$

$$Y f \longmapsto f(Y f).$$

This is called the <u>Y-combinator</u>.

Paul Downen (Lect 2)

The Y-combinator lets us introduce recursive functions into our language.

A recursive function is a fixed point.

EX: (times)

$$\text{times} = \lambda x\, \lambda y.\ \text{if } x=0 \text{ then } 0 \text{ else } y + (\text{times}\ (x-1)\ y)$$

It would seem we can't do this directly in λ-calc because times calls itself.

However, we can do

$$\text{timesish} := \lambda \text{next}.\, \lambda x.\, \lambda y.\ \text{if } x=0 \text{ then } 0 \text{ else } y + (\text{next}\ x-1\ y)$$

idea: the 1st argument 'next' says how to take a step.

Define:

$$\text{times} = Y\ \text{timesish}$$

... then times is a fixed point of timesish; i.e.

$$\text{timesish}\ (Y\ \text{timesish}) = Y\ \text{timesish}$$

Paul Downen | Lect 2 [2018/06/03]

Simply Typed Lambda Calculus

$\tau \in Type ::= \tau_1 \to \tau_2 \mid \alpha$

$$\frac{}{\Gamma, x:\tau \vdash x:\tau} \qquad \frac{\Gamma \vdash e:\tau' \to \tau \quad \Gamma \vdash e':\tau'}{\Gamma \vdash e\,e' : \tau}$$

$$\frac{\Gamma, x:\tau \vdash e:\tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

Theorem (termination)

If $\Gamma \vdash e:\tau$ then there is an e' such that $e \mapsto^* e' \not\mapsto$