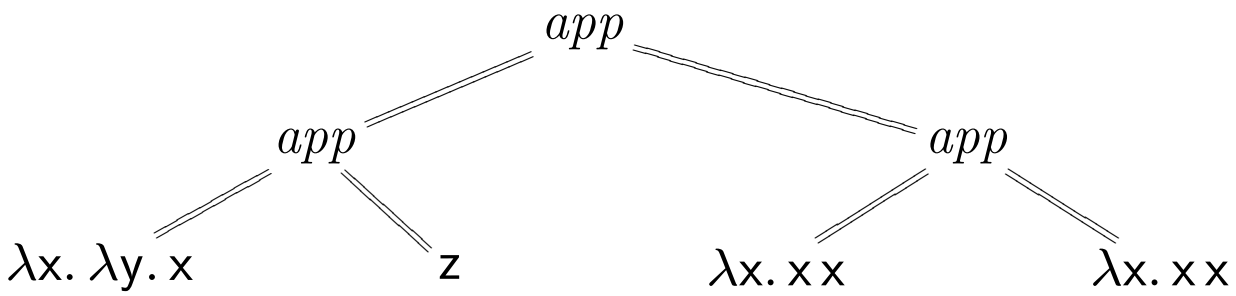
Overview

- Normal-order reduction of lambda terms
- η-reduction
- Programming in the lambda calculus: Church numerals
- Normal-order evaluation

Normal-Order Reduction

$K\,\mathsf{z}\,\Omega \mapsto (\lambda \mathsf{y}.\,\mathsf{z})\,\Omega \mapsto \mathsf{z}$ $\qquad$ $K\,\mathsf{z}\,\Omega \mapsto K\,\mathsf{z}\,\Omega \mapsto \ldots$



An outermost redex is one not contained in any other redex.

In the normal-order reduction sequence of a term

at each step the contracted redex is the leftmost outermost one.

$$(\lambda \mathsf{y}.\,\mathsf{y}\,\mathsf{y}\,((\lambda \mathsf{x}.\,\mathsf{x}\,\mathsf{x})\,\Delta))\,K \mapsto K\,K$$

Theorem [Standardization]:
If e has a normal form,

then the normal-order reduction sequence starting with e terminates.

η-Reduction

For every $e \in exp$,

the terms e and $\lambda v.\, e\, v$ (where $v \notin FV(e)$) are extensionally equivalent — they reduce to the same term when applied to any other term e':

$$(\lambda v.\, e\, v)\, e' \mapsto e\, e'$$

Hence the η-reduction rule:

$$\frac{}{\lambda v.\, e\, v \mapsto e} \text{ when } v \notin FV(e)$$

The Church-Rosser and Standardization properties hold for the $\beta\eta$-reduction (the union of β- and η-reduction).

Programming in the Lambda Calculus

Idea: Encode data as combinators in normal-form
— then uniqueness of normal form then guarantees we can decode a valid result.

Example: Church numerals (note that NUM_n is in normal form for every $n \in \mathbf{N}$)

$$
\begin{array}{rcl}
NUM_n & \stackrel{\text{def}}{=} & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, P_n \\
\text{where } P_0 & = & \mathsf{x} \\
P_{n+1} & = & \mathsf{f}\, P_n \\
\text{i.e. } P_n & = & \underbrace{\mathsf{f}\,(\ldots(\mathsf{f}}_{n \text{ times}}\,\mathsf{x})\ldots) \\
SUCC & \stackrel{\text{def}}{=} & \lambda \mathsf{n}.\, \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\,(\mathsf{n}\,\mathsf{f}\,\mathsf{x}) \\
\\
SUCC\ NUM_n & = & (\lambda \mathsf{n}.\, \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\,(\mathsf{n}\,\mathsf{f}\,\mathsf{x}))\,(\lambda \mathsf{f}.\, \lambda \mathsf{x}.\, P_n) \\
 & \mapsto & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\,((\lambda \mathsf{f}.\, \lambda \mathsf{x}.\, P_n)\,\mathsf{f}\,\mathsf{x}) \\
 & \mapsto & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\,((\lambda \mathsf{x}.\, P_n)\,\mathsf{x}) \\
 & \mapsto & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\, P_n \\
 & \mapsto & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, P_{n+1} \\
 & = & NUM_{n+1}
\end{array}
$$

Programming with Church Numerals

In Haskell one could implement addition and multiplication using recursion:

```
data Num = Zero | Succ Num

add Zero     n = n
add (Succ m) n = Succ (add m n)
mul Zero     n = Zero
mul (Succ m) n = add n (mul m n)
```

Recursion can also be encoded in lambda calculus,
but one can avoid recursion and use Church numerals as iterators:

$$ADD \stackrel{\text{def}}{=} \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\, f\, (n\, f\, x) \qquad ADD\, NUM_m\, NUM_n \mapsto^* NUM_{m+n}$$

$$MUL \stackrel{\text{def}}{=} \lambda m.\, \lambda n.\, \lambda f.\, m\, (n\, f) \qquad MUL\, NUM_m\, NUM_n \mapsto^* NUM_{mn}$$

$$EXP \stackrel{\text{def}}{=} \lambda m.\, \lambda n.\, n\, m \qquad EXP\, NUM_m\, NUM_n \mapsto^* NUM_{m^n}$$

Addition of Church Numerals

$$
\begin{array}{rcl}
ADD & \stackrel{\text{def}}{=} & \lambda \mathsf{m}.\, \lambda \mathsf{n}.\, \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{m}\, \mathsf{f}\, (\mathsf{n}\, \mathsf{f}\, \mathsf{x}) \\
 & & \\
ADD\, NUM_m\, NUM_n & \mapsto^2 & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, NUM_m\, \mathsf{f}\, (NUM_n\, \mathsf{f}\, \mathsf{x}) \\
 & \mapsto^2 & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, NUM_m\, \mathsf{f}\, P_n \\
 & \mapsto^2 & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, (P_m/\mathsf{x} \to P_n) \\
 & = & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \underbrace{\mathsf{f}\, (\ldots (\mathsf{f}}_{m \text{ times}}\ P_n) \ldots) \\
 & = & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \underbrace{\mathsf{f}\, (\ldots (\mathsf{f}}_{m \text{ times}}\, (\underbrace{\mathsf{f}\, (\ldots (\mathsf{f}}_{n \text{ times}}\, \mathsf{x}) \ldots))) \ldots) \\
 & = & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, P_{m+n} \\
 & = & NUM_{m+n}
\end{array}
$$

Normal-Order Evaluation

- Canonical form: a term with no "top-level" redexes;
  in the pure lambda calculus: an abstraction.

  A typical functional programming language allows functions
  to only be applied but not inspected,
  so once a result is known to be a function, it is not reduced further.

- Evaluation: reduction of closed expressions.

  A typical programming language defines programs as closed terms.
  If e is closed, $e \Rightarrow z$ (e evaluates to z) when z is the first canonical form
  in the normal-order reduction sequence of e.

- Even if the normal-order reduction sequence is infinite,
  it may contain a canonical form;

  however other reduction sequences may contain other canonical forms.

Reduction of Closed Terms

A closed term e either diverges or reduces to a canonical form.

Proof:
Reduction does not introduce free variables, hence every term of the sequence is closed.

If the sequence is finite, it ends with a normal form which can only be an abstraction:

By induction, a closed normal form can only be an abstraction:

- A variable v is a normal form but not a closed term;
- An application $e_1\ e_2$ can be a normal form only if e_1 is a normal form;

  but then by IH e_1, being a closed normal form, may only be an abstraction;

  then $e_1\ e_2$ is a redex,

  hence is not a normal form, contradicting the assumption.

Big-Step (Natural) Semantics for Normal-Order Evaluation

Inference rules for evaluation:

$$\text{(termination)} \qquad \frac{}{\vdash \lambda v.\, e \Rightarrow \lambda v.\, e}$$

$$(\beta\text{-evaluation}) \qquad \frac{\vdash e \Rightarrow \lambda v.\, e_1 \qquad \vdash (e_1/v \rightarrow e') \Rightarrow z}{\vdash e\, e' \Rightarrow z}$$

Proposition:
If e is a closed term and z is a canonical form, $e \Rightarrow z$ if and only if $\vdash e \Rightarrow z$ is provable.

Hence the recursive algorithm for normal-order evaluation of a closed e is:

- if e is an abstraction, it evaluates to e;
- otherwise $e = e_1\, e_2$; first evaluate e_1 to its canonical form, an abstraction $\lambda v.\, e_1'$, then the value of e is that of $e_1'/v \rightarrow e_2$.

Examples of Normal-Order Evaluation

- Expression diverging under $\beta\eta$-reduction may have canonical forms:

$$K\,\Omega = (\lambda x.\,\lambda y.\,x)\,\Omega \mapsto \lambda y.\,\Omega$$

  hence $K\,\Omega \Rightarrow \lambda y.\,\Omega$, although $(K\,\Omega)$ diverges under $\beta\eta$-reduction.

- Expressions with the same $\beta\eta$-normal form may have different canonical forms under normal order evaluation:

$$(\lambda x.\,\lambda y.\,x)\,\Delta \mapsto^* \lambda y.\,\Delta \qquad\qquad (\lambda x.\,\lambda y.\,x)\,\Delta \Rightarrow \lambda y.\,\Delta$$

$$\lambda x.\,(\lambda y.\,y)\,\Delta \mapsto^* \lambda x.\,\Delta \qquad \text{but} \qquad \lambda x.\,(\lambda y.\,y)\,\Delta \Rightarrow \lambda x.\,(\lambda y.\,y)\,\Delta$$

- Sometimes normal order evaluation performs more work
  (i.e. needs more steps to get to the same term)
  than other reduction orders:

normal order:	$\Delta\,(I\,I)$	$\mapsto (I\,I)\,(I\,I)$	$\mapsto I\,(I\,I)$	$\mapsto I\,I$	$\mapsto I$
another (eager) strategy:	$\Delta\,(I\,I)$	$\mapsto \Delta\,I$	$\mapsto I\,I$	$\mapsto I$	