The Lambda Calculus

- The Greatest Thing Since Sliced Bread™, or maybe even before it
- The basis of functional languages (ML, Haskell, LISP, Algol 60…)
- Close connection with logic:
  - Developed by logicians Church, Rosser, Curry since 1930s
  - Intended as a formal proof notation
    allowing proof transformation
- Easier to reason about than procedural imperative languages:
  has no assignment operation and needs no state in its semantics —
  all computation is expressed as applications of abstractions

The Pure Untyped Lambda Calculus

Syntax:

$$
\begin{array}{rcll}
exp & ::= & var & \text{variable} \\
 & | & \lambda var.\, exp & \text{abstraction (lambda expression)} \\
 & | & exp\ exp & \text{application}
\end{array}
$$

Conventions:

- the body e of the abstraction $\lambda v.\, e$ extends as far as the syntax allows — to a closing parenthesis or end of term
- application is left associative

$$\lambda \mathsf{x}.\, (\lambda \mathsf{y}.\, \mathsf{x}\,\mathsf{y}\,\mathsf{y})\lambda \mathsf{x}.\, \lambda \mathsf{z}.\, \mathsf{x}\,\mathsf{z} \;\equiv\; \lambda \mathsf{x}.\, ((\lambda \mathsf{y}.\, ((\mathsf{x}\,\mathsf{y})\,\mathsf{y}))(\lambda \mathsf{x}.\, (\lambda \mathsf{z}.\, (\mathsf{x}\,\mathsf{z}))))$$

Syntactic Properties

$\lambda v.\, e$ binds v in e, so we define the free variables of a lambda term by

$$
\begin{aligned}
FV(v) &= \{v\} \\
FV(e\, e') &= FV(e) \cup FV(e') \\
FV(\lambda v.\, e) &= FV(e) - \{v\}
\end{aligned}
$$

and substitution as

$$
\begin{aligned}
v/\delta &= \delta\, v \\
(e\, e')/\delta &= (e/\delta)\,(e'/\delta) \\
(\lambda v.\, e)/\delta &= \lambda v_{\mathsf{new}}.\,(e/[\delta \mid v : v_{\mathsf{new}}]) \\
\text{where } v_{\mathsf{new}} &\notin \bigcup_{w \in FV(e) - \{v\}} FV(\delta\, w)
\end{aligned}
$$

Renaming of bound variables

Renaming of bound variables:

Replacing $\lambda v.\, e$ with $\lambda v'.\, (e/v \rightarrow v')$ where $v' \notin FV(e) - \{v\}$.

e' is α-equivalent to e ($e \equiv e'$)

if it is obtained from e by renaming of bound variables of subterms.

The semantics of lambda calculus identifies all α-equivalent terms.

Reduction

The first semantics of the lambda calculus was operational,

based on a notion of reduction on terms.

Configurations are lambda terms: $\Gamma = exp$.

The single step relation $\mapsto$ is not a function:

reduction is nondeterministic

but a terminal configuration, if it exists, is unique.

The central rule is for β-reduction (β-contraction):

$$\frac{}{(\lambda v.\, e)\, e' \;\mapsto\; e/v \to e'}$$

$(\lambda v.\, e)\, e'$ is the redex and $e/v \to e'$ is its contractum.

Contextual Rules for Reduction

$$\frac{e \mapsto e'}{\lambda v.\, e \mapsto \lambda v.\, e'} \qquad \frac{e \equiv e'}{e \mapsto^* e'}$$

$$\frac{e_0 \mapsto e_0'}{e_0\, e_1 \mapsto e_0'\, e_1} \qquad \frac{e \mapsto e'}{e \mapsto^* e'}$$

$$\frac{e_1 \mapsto e_1'}{e_0\, e_1 \mapsto e_0\, e_1'} \qquad \frac{e_0 \mapsto^* e_1 \qquad e_1 \mapsto^* e_2}{e_0 \mapsto^* e_2}$$

$$\frac{e_0 \mapsto e_0' \qquad e_0' \equiv e_1'}{e_0 \mapsto e_1'}$$

An expression containing no redexes is a (or in) normal form;

normal forms correspond to terminal configurations.

An expression e has a normal form

if $\exists e' \in exp.\ e \mapsto^* e'$ and e' is a normal form.

Confluence: The Church-Rosser Theorem

The single-step reduction is nondeterministic,
but determinism is eventually recovered in the interesting cases:

Theorem [Church-Rosser]:
For all $e,\ e_0,\ e_1 \in exp$,
if $e \mapsto^* e_0$ and $e \mapsto^* e_1$, then there exists $e' \in exp$ such that $e_0 \mapsto^* e'$ and $e_1 \mapsto^* e'$.

Corollary:
Every expression has at most one normal from (up to α-equivalence).

Proof:
If $e \mapsto^* e_0$ and $e \mapsto^* e_1$ and both e_0 and e_1 are normal forms,
then by Church-Rosser there is some e' such that $e_0 \mapsto^* e'$ and $e_1 \mapsto^* e'$.
But neither e_0 nor e_1 have redexes,
so the only rule that can be applied to them is that of α-equivalence.

Examples of Reduction

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

$I \stackrel{\text{def}}{=} \lambda \mathsf{x}.\, \mathsf{x}$ is the identity combinator

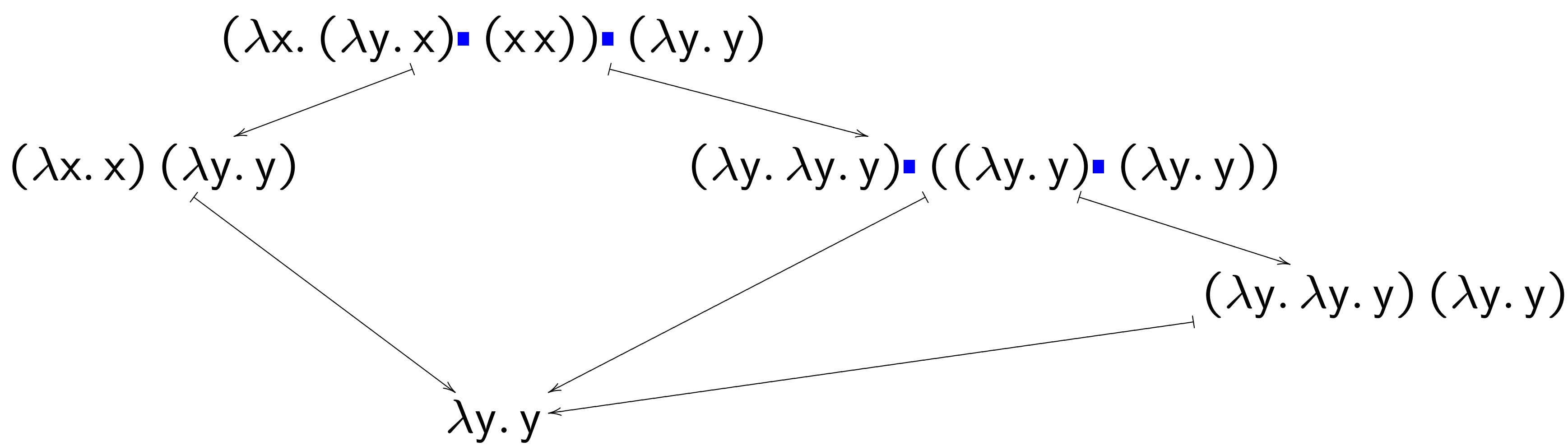
(combinator = closed term)

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

$K \stackrel{\text{def}}{=} \lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathsf{x}$ is the constant combinator

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

$\Delta \stackrel{\text{def}}{=} \lambda \mathsf{x}.\, \mathsf{x}\, \mathsf{x}$ is the self-application comb.



Normal-Order Reduction

$\Delta\,\Delta \equiv (\lambda \mathsf{x}.\, \mathsf{x}\,\mathsf{x})\,\Delta \mapsto \Delta\,\Delta \mapsto \ldots$ $\qquad \Omega \stackrel{\text{def}}{=} \Delta\,\Delta$ is a diverging expression

$K\,\mathsf{z}\,\Omega \mapsto K\,\mathsf{z}\,\Omega \mapsto \ldots,$ $\qquad$ (where $K = \lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathsf{x}$), but also

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

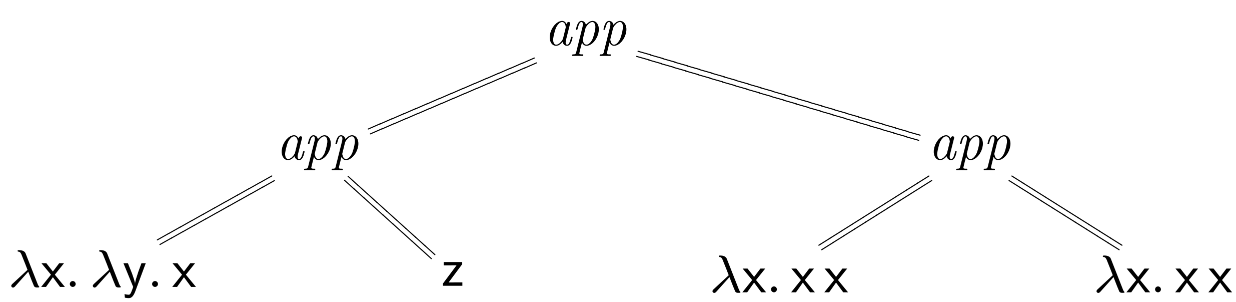
An outermost redex is one not contained in any other redex.

In the normal-order reduction sequence of a term
at each step the contracted redex is the leftmost outermost one.

Theorem [Standardization]:
If e has a normal form,
then the normal-order reduction sequence starting with e terminates.

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 $n \in \mathbf{N}$)

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

$$SUCC \stackrel{\text{def}}{=} \lambda \mathsf{n}.\, \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\, (\mathsf{n}\, \mathsf{f}\, \mathsf{x})$$

$$\begin{array}{rcl}
SUCC\ NUM_n & = & (\lambda \mathsf{n}.\, \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\, (\mathsf{n}\, \mathsf{f}\, \mathsf{x}))\, (\lambda \mathsf{f}.\, \lambda \mathsf{x}.\, P_n) \\
& \mapsto & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\, ((\lambda \mathsf{f}.\, \lambda \mathsf{x}.\, P_n)\, \mathsf{f}\, \mathsf{x}) \\
& \mapsto & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\, ((\lambda \mathsf{x}.\, P_n)\, \mathsf{x}) \\
& \mapsto & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{f}\, P_n \\
& \mapsto & \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, P_{n+1} \\
& = & NUM_{n+1}
\end{array}$$

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:

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

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

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

Addition of Church Numerals

$$ADD \stackrel{\text{def}}{=} \lambda \mathsf{m}.\, \lambda \mathsf{n}.\, \lambda \mathsf{f}.\, \lambda \mathsf{x}.\, \mathsf{m}\, \mathsf{f}\, (\mathsf{n}\, \mathsf{f}\, \mathsf{x})$$

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

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:

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

hence $K\,\Omega \Rightarrow \lambda \mathsf{y}.\,\Omega$, 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:

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

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

- 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:	$\Delta\,(I\,I)$	$\mapsto \Delta\,I$	$\mapsto I\,I$	$\mapsto I$	

Eager Evaluation

Sometimes normal order evaluation performs more work than other reduction orders:

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

Reason: When reducing $(\lambda v.\, e)\, e' \mapsto e/v \to e'$,
redexes in e' are replicated in $e/v \to e'$ if v occurs more than once in e.

Solution: Use eager evaluation order:
First evaluate the argument e' to canonical form.

β_E-reduction rule:

$$\frac{}{(\lambda v.\, e)\, z \mapsto (e/v \to z)} \qquad \text{if } z \text{ is a canonical form or a variable}$$

$e \Rightarrow_E z$ (e evaluates eagerly to z)
if there is a reduction sequence from e to z
of contractions of the leftmost β_E-redexes not inside a canonical form.

Inference Rules for Eager Evaluation

(termination)
$$\frac{}{\vdash \lambda v.\, e \Rightarrow_E \lambda v.\, e}$$

(β_E-evaluation)
$$\frac{\vdash e_1 \Rightarrow_E \lambda v.\, e_1' \qquad \vdash e_2 \Rightarrow_E z_2 \qquad \vdash (e_1'/v \to z_2) \Rightarrow_E z}{\vdash e_1\, e_2 \Rightarrow_E z}$$

This is the strategy used by most implementations of "strict" languages.

Recursive algorithm for eager evaluation of a closed e:

- 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 evaluate e_2 to a canonical form z_2,
  - then the value of e is that of $e_1'/v \to z_2$.

Eager evaluation performs more work than normal-order evaluation when the parameter does not occur in the abstraction body:

$$(\lambda \mathrm{x}.\, I)\, \Omega \not\Rightarrow_E \qquad \text{but} \qquad (\lambda \mathrm{x}.\, I)\, \Omega \Rightarrow I$$

Denotational Semantics of the Lambda Calculus

We need a set S of denotations and a meaning function $[\![-]\!]$ such that $[\![-]\!] \in exp \to S$, and a lambda calculus application is interpreted as a function application:

$$[\![e\, e']\!] = [\![e]\!]\, [\![e']\!]$$

So the set S must contain functions from S to S.

If S contains all functions from S to S, the problem has no non-trivial solutions due to Russell's paradox:

- If $S \to S \subseteq S$, we can construct a fixed point of every function $f \in S \to S$:
  Let $p = \lambda x \in S. \begin{cases} f\,(x\,x), & \text{if } x \in S \to S \\ x \text{ (or anything else in } S\text{)}, & \text{otherwise.} \end{cases}$
  Then $p\,p = f\,(p\,p)$ is a fixed point of f.
  The lambda term that manifests this construction is a fixed-point combinator Y:

  $$Y \stackrel{\text{def}}{=} \lambda \mathsf{f}.\, (\lambda \mathsf{x}.\, \mathsf{f}\,(\mathsf{x}\,\mathsf{x}))\, \lambda \mathsf{x}.\, \mathsf{f}\,(\mathsf{x}\,\mathsf{x})$$
  $$Y\, e \mapsto (\lambda \mathsf{x}.\, e\,(\mathsf{x}\,\mathsf{x}))\, \lambda \mathsf{x}.\, e\,(\mathsf{x}\,\mathsf{x}) \mapsto e\,((\lambda \mathsf{x}.\, e\,(\mathsf{x}\,\mathsf{x}))\, \lambda \mathsf{x}.\, e\,(\mathsf{x}\,\mathsf{x}))$$

- But if S has more than one element, not all functions in $S \to S$ have fixed points, e.g. $not \in \mathbf{B} \to \mathbf{B}$: no element $b \in \mathbf{B}$ satisfies $b = not\, b$.

Scott's Recursive Domain Isomorphism for the Lambda Calculus

Dana Scott solved the problem by considering a domain of values and requiring the functions in it to be continuous.

Scott's Domain D_∞ satisfies the isomorphism

$$D_\infty \underset{\psi}{\overset{\phi}{\rightleftarrows}} [D_\infty \to D_\infty]$$

Then the meaning of a lambda calculus term can be given by a function

$$[\![-]\!] \in exp \to [Env \to D_\infty]$$

where $Env \stackrel{\text{def}}{=} var{\to}D_\infty$ is the set of environments assigning values to free variables.

Semantic Equations

$$D_\infty \underset{\psi}{\overset{\phi}{\rightleftarrows}} [D_\infty \to D_\infty]$$

$$\begin{aligned}
[\![-]\!] &\in exp \to [(var \to D_\infty) \to D_\infty] \\
[\![v]\!]\,\eta &= \eta\, v \\
[\![\lambda v.\, e]\!]\,\eta &= \psi\,(\lambda x \in D_\infty.\, [\![e]\!][\eta \,|\, v : x]) \\
[\![e\, e']\!]\,\eta &= \phi\,([\![e]\!]\,\eta)\,([\![e']\!]\,\eta)
\end{aligned}$$

We have to prove that all terms in this definition are in the required domains:

- $\lambda x \in D_\infty.\, [\![e]\!][\eta \,|\, v : x]$ is a continuous function from D_∞ to D_∞
- the so-defined $[\![-]\!]$ is a continuous function from Env to D_∞.

Correctness of the Semantic Equations

Using the continuous (for any predomains $P,\ P',\ P''$) functions

$$\begin{array}{rcl} get_{P,v}\,\eta = \eta\, v & & get_{P,v} \in [(var \to P) \to P] \\ ext_{P,v}\,\langle \eta,\, x\rangle = [\eta \mid v : x] & & ext_{P,v} \in [(var \to P) \times P \to var \to P] \\ ap_{P,P'}\,\langle f,\, x\rangle = f\, x & & ap_{P,P'} \in [(P \to P') \times P \to P'] \\ ((ab_{P,P',P''}\, f)\, x)\, y = f\,\langle x,\, y\rangle & & ab_{P,P',P''} \in [P \times P' \to P''] \to [P \to [P' \to P'']] \end{array}$$

rewrite the semantic equations:

$$\begin{array}{rcl} [\![v]\!] = \lambda\eta \in Env.\, \eta\, v & & = get_{D_\infty,v} \\ [\![\lambda v.\, e]\!] = \lambda\eta \in Env.\, \psi\,(\lambda x \in D_\infty.\, [\![e]\!][\eta \mid v : x]) & & = \psi \cdot ab_{Env,D_\infty,D_\infty}\,([\![e]\!] \cdot ext_{D_\infty,v}) \\ [\![e\, e']\!] = \lambda\eta \in Env.\, \phi\,([\![e]\!]\,\eta)\,([\![e']\!]\,\eta) & & = ap_{D_\infty,D_\infty} \cdot ((\phi \cdot [\![e]\!]) \otimes [\![e']\!]) \end{array}$$

Well-formedness and continuity of $[\![-]\!]$ follows from continuity of $\cdot$ and $\otimes$.

Properties of the Denotational Semantics

Coincidence: If $\forall v \in FV(e).\ \eta\, v = \eta'\, v$, then $[\![e]\!]\,\eta = [\![e]\!]\,\eta'$.

Substitution: If $\forall v \in FV(e).\ [\![\delta v]\!]\,\eta' = \eta\, v$, then $[\![e/\delta]\!]\,\eta' = [\![e]\!]\,\eta$.

Finite Substitution:

$$[\![e/v_1 \to e_1, \ldots\ v_n \to e_n]\!]\,\eta = [\![e]\!]\,[\eta \mid v_1 : [\![e_1]\!]\,\eta \mid \ \ldots\ \mid v_n : [\![e_n]\!]\,\eta].$$

Renaming Preserves Meaning: (i.e. α-equivalence is sound w.r.t. the semantics)

If $w \notin FV(e) - \{v\}$, then $[\![\lambda w.\,(e/v \to w)]\!] = [\![\lambda v.\,e]\!]$.

Soundness of β-contraction: $[\![(\lambda v.\,e)\,e']\!] = [\![e/v \to e']\!]$

(from $\phi \cdot \psi = I_{[D_\infty \to D_\infty]}$)

Soundness of η-contraction: If $v \notin FV(e)$, then $[\![\lambda v.\,e\,v]\!] = [\![e]\!]$

(from $\psi \cdot \phi = I_{D_\infty}$)

Soundness of β-Contraction

For any $\eta \in Env$,

$$
\begin{aligned}
[\![(\lambda v.\, e)\, e']\!]\, \eta &= \phi\, ([\![\lambda v.\, e]\!]\, \eta)\, ([\![e']\!]\, \eta) && \text{semantics of application} \\
&= \phi\, (\psi\, (\lambda x \in D_\infty.\, [\![e]\!]\, [\eta \mid v : x]))\, ([\![e']\!]\, \eta) && \text{semantics of abstraction} \\
&= (\lambda x \in D_\infty.\, [\![e]\!]\, [\eta \mid v : x])\, ([\![e']\!]\, \eta) && \text{half of the isomorphism} \\
&= [\![e]\!]\, [\eta \mid v : [\![e']\!]\, \eta] && \\
&= [\![e/v \to e']\!]\, \eta && \text{finite substitution}
\end{aligned}
$$

Soundness of η-Contraction

For any $\eta \in Env$,

$$
\begin{aligned}
[\![\lambda v.\, e\, v]\!]\, \eta &= \psi\, (\lambda x \in D_\infty.\, [\![e\, v]\!]\, [\eta \mid v : x]) && \text{semantics of abstraction} \\
&= \psi\, (\lambda x \in D_\infty.\, \phi\, ([\![e]\!]\, [\eta \mid v : x])\, ([\![v]\!]\, [\eta \mid v : x])) && \text{semantics of application} \\
&= \psi\, (\lambda x \in D_\infty.\, \phi\, ([\![e]\!]\, \eta)\, ([\![v]\!]\, [\eta \mid v : x])) && \text{coincidence} \\
&= \psi\, (\lambda x \in D_\infty.\, \phi\, ([\![e]\!]\, \eta)\, x) && \text{semantics of a variable} \\
&= \psi\, (\phi\, ([\![e]\!]\, \eta)) && \\
&= [\![e]\!]\, \eta && \text{half of the isomorphism}
\end{aligned}
$$

The Least Fixed-Point Combinator

The fixed-point combinator

$$Y = \lambda \mathsf{f}.\, (\lambda \mathsf{x}.\, \mathsf{f}\,(\mathsf{x}\,\mathsf{x}))\, \lambda \mathsf{x}.\, \mathsf{f}\,(\mathsf{x}\,\mathsf{x})$$

denotes (up to isomorphism)

the least fixed-point operator on the Scott's Domain D_∞:

$$[\![Y]\!]\eta = \psi\,(\mathbf{Y}_{D_\infty} \cdot \phi)$$

Semantics of Normal-Order Evaluation

In the given semantics $[\![\lambda \mathrm{x}.\, \Omega]\!] = \bot$ (the term diverges under $\beta\eta$-reduction).

But $\lambda \mathrm{x}.\, \Omega$ is a canonical form under normal-order evaluation (NOE),
so its denotation must be different from $\bot = [\![\Omega]\!]$.

$\Rightarrow$ the semantic domain D for NOE must include a least element $\bot$
in addition to a set of values of canonical forms V isomorphic to $[D \to D]$:

$$D = V_\bot \;\text{ where }\; V \cong [D \to D]$$

If $V \underset{\psi}{\overset{\phi}{\rightleftarrows}} [D \to D]$, then $D \underset{\iota_\uparrow \cdot \psi}{\overset{\phi_{\perp\!\!\!\perp}}{\rightleftarrows}} [D \to D]$, but the latter is not an isomorphism.
The semantic equations then are similar but using the new pair $\phi_{\perp\!\!\!\perp}$ and $\iota_\uparrow \cdot \psi$:

$$\begin{aligned} [\![v]\!]\,\eta &= \eta\, v \\ [\![\lambda v.\, e]\!]\,\eta &= (\iota_\uparrow \cdot \psi)\,(\lambda x \in D_\infty.\, [\![e]\!][\eta \mid v : x]) \\ [\![e\, e']\!]\,\eta &= \phi_{\perp\!\!\!\perp}\,([\![e]\!]\,\eta)\,([\![e']\!]\,\eta) \end{aligned}$$

Normal-Order Evaluation and η-Contraction

In this semantics $\phi_{\perp\!\!\perp}$ and $\iota_{\uparrow} \cdot \psi$ do not define an isomorphism between D and $[D \to D]$:

$$\phi_{\perp\!\!\perp} \cdot (\iota_{\uparrow} \cdot \psi) = I_{[D \to D]}, \qquad \text{but} \qquad (\iota_{\uparrow} \cdot \psi) \cdot \phi_{\perp\!\!\perp} \neq I_D$$

Hence β-reduction is sound, while η-reduction is not.

Just what we expected:

$\lambda \mathsf{x}.\, \Omega\, \mathsf{x} \Rightarrow \lambda \mathsf{x}.\, \Omega\, \mathsf{x}$ is a canonical form, but $\lambda \mathsf{x}.\, \Omega\, \mathsf{x} \overset{\eta}{\mapsto} \Omega \not\Rightarrow$,

so $[\![\lambda \mathsf{x}.\, \Omega\, \mathsf{x}]\!] \neq \Omega$ under NOE.

Y again corresponds to the least fixed fixed-point operator on D.

Semantics of Eager Evaluation

Under eager evaluation arguments are reduced to canonical forms first,

so denotations of functions only operate on values in V

$\Rightarrow$ the environments are in $[var \to V]$, and the domain equation is

$$D = V_\perp \text{ where } V \cong [V \to D]$$

If $V \underset{\psi}{\overset{\phi}{\rightleftarrows}} [V \to D]$, the semantic equations are

$$\begin{aligned}
[\![v]\!]\,\eta &= \iota_\uparrow\,(\eta\, v) \\
[\![\lambda v.\, e]\!]\,\eta &= (\iota_\uparrow \cdot \psi)\,(\lambda x \in D_\infty.\, [\![e]\!][\eta \,|\, v : x]) \\
[\![e\, e']\!]\,\eta &= (\phi_{\perp\!\!\!\perp}\,([\![e]\!]\,\eta))_{\perp\!\!\!\perp}\,([\![e']\!]\,\eta)
\end{aligned}$$

Note: $\iota_\uparrow$ is used to inject into D values from V

(denotations of canonical forms).

The Fixed-Point Combinator Y and Eager Evaluation

The fixed-point combinator $Y = \lambda \mathsf{f}.\, (\lambda \mathsf{x}.\, \mathsf{f}\, (\mathsf{x}\, \mathsf{x}))\, \lambda \mathsf{x}.\, \mathsf{f}\, (\mathsf{x}\, \mathsf{x})$

is not suitable for eager evaluation because $Y\ e$ diverges for any e:

$$
\begin{array}{lll}
Y\ e & \overset{\beta}{\longmapsto}{}^{*}_{E} & Y\ z \qquad\qquad \text{if } e \Rightarrow_E z \\
 & \overset{\beta}{\longmapsto}_{E} & (\lambda v.\, z\, (v\, v))\, \lambda v.\, z\, (v\, v) \qquad\qquad \text{where } v \notin FV(z) \\
 & \overset{\beta}{\longmapsto}_{E} & z\, ((\lambda v.\, z\, (v\, v))\, \lambda v.\, z\, (v\, v)) \\
 & \overset{\beta}{\longmapsto}_{E} & z\, (z\, ((\lambda v.\, z\, (v\, v))\, \lambda v.\, z\, (v\, v))) \\
 & \overset{\beta}{\longmapsto}_{E} & \ldots
\end{array}
$$

The Fixed-Point Combinator Y_v

Instead, use the call-by-value fixed-point combinator

$$Y_v \stackrel{\text{def}}{=} \lambda \mathsf{f}.\,(\lambda \mathsf{x}.\,\mathsf{f}\,(\lambda \mathsf{y}.\,\mathsf{x}\,\mathsf{x}\,\mathsf{y}))\,\lambda \mathsf{x}.\,\mathsf{f}\,(\lambda \mathsf{y}.\,\mathsf{x}\,\mathsf{x}\,\mathsf{y})$$

For any term e, if $e \Rightarrow_E z$, then

$$Y_v\,e \;\overset{\beta}{\longmapsto}{}^{*}_{E}\; e' \stackrel{\text{def}}{=} (\lambda v.\,z\,(\lambda \mathsf{y}.\,v\,v\,\mathsf{y}))\,\lambda v.\,z\,(\lambda \mathsf{y}.\,v\,v\,\mathsf{y}) \qquad \text{where } v \notin FV(z)$$

such that $\lambda v.\,e'\,v$ is extensionally a fixed-point of e: for any term e_1,

$$\begin{aligned}
(\lambda v.\,e'\,v)\,e_1 \;&\overset{\beta}{\longmapsto}_E\; e'\,e_1 \\
&\overset{\beta}{\longmapsto}_E\; z\,(\lambda v.\,(\lambda v.\,z\,(\lambda \mathsf{y}.\,v\,v\,\mathsf{y}))\,(\lambda v.\,z\,(\lambda \mathsf{y}.\,v\,v\,\mathsf{y}))\,v)\,e_1 \\
&=\; z\,(\lambda v.\,e'\,v)\,e_1
\end{aligned}$$

$$(e\,(\lambda v.\,e'\,v))\,e_1 \;\overset{\beta}{\longmapsto}{}^{*}_{E}\; z\,(\lambda v.\,e'\,v)\,e_1$$