

Lambda calculus

- A starting point for reasoning about functions
- The foundation of most functional programming languages, including the Lisp and ML languages

v ranges over a countable number of variables

e	$::=$	v	(variables)
		$e_1 e_2$	(function application)
		$\lambda v.e$	(function abstraction)



Evaluation

$$\lambda f.f \ 1 \quad \rightarrow \quad \lambda f.f \ 1$$

$$(\lambda f.f) \ 1 \quad \rightarrow \quad 1$$

$$\begin{aligned} (\lambda f.f \ 1) (\lambda x.x + 1) &\rightarrow (\lambda x.x + 1) \ 1 \\ &\rightarrow 1 + 1 \\ &\rightarrow 2 \end{aligned}$$

$$\begin{aligned} (\lambda x.x \ x) (\lambda y.y \ y) &\rightarrow (\lambda y.y \ y) (\lambda y.y \ y) \\ &\rightarrow (\lambda y.y \ y) (\lambda y.y \ y) \\ &\rightarrow (\lambda y.y \ y) (\lambda y.y \ y) \end{aligned}$$

⋮



Reduction

- Program evaluation uses a substitution semantics.
- ``To evaluate an application $(\lambda v.e_1) e_2$, substitute e_2 for v in e_1 .''
- Substitution: $e_1[e_2/v]$ means:
 - Substitute e_2 for v in e_1
 - Replace all occurrences of v in e_1 with e_2
- Sometimes written $e_1[v := e_2]$.



Single-step evaluation

- A single-step reduction is just a substitution
- Called “beta-reduction”

$$(\lambda v. e_1) e_2 \rightarrow_{\beta} e_1[e_2/v]$$



Defining substitution

- Define substitution $e_1[e_2/v]$ by induction on the structure of e_1
 - e_1 is a variable v' (v' may or may not be the same variable as v)
 - e_1 is a function application $e_3 e_4$
 - e_1 is a function abstraction $\lambda v'.e_3$ (v' may or may not be the same variable as v)



Capture-avoiding substitution

- $v'[e_2/v] = \begin{cases} e_2 & \text{if } v = v' \\ v' & \text{otherwise} \end{cases}$
- $(e_3 e_4)[e_2/v] = (e_3[e_2/v] e_4[e_2/v])$
- $(\lambda v'.e_3)[e_2/v] = \begin{cases} \lambda v'.e_3 & \text{if } v = v' \\ \lambda v'.e_3[e_2/v] & \text{if } v' \notin FV(e_2) \\ \lambda v''.e_3[v''/v'] [e_2/v] & \text{otherwise, new } v'' \end{cases}$



Variables

- An occurrence of a variable is a “binding occurrence” if it is defined by a lambda
- An occurrence is “bound” if it is in the scope of a binding occurrence with the same name
- Other occurrences are “free”



Free variables

Free variables are defined by induction (as usual):

$$\begin{aligned}FV(v) &= \{v\} \\FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \\FV(\lambda v.e) &= FV(e) - \{v\} \\FV(\lambda x.\lambda y.x) &= \{\} \\FV(\lambda x.x + y) &= \{y\} \\FV(\lambda x.(\lambda y.x) (\lambda z.y)) &= \{y\}\end{aligned}$$



Some sensible properties

- Any two alpha-equivalent terms have the same free variables
- Free variables only decrease during evaluation
- Beta-reduction works:

If $e_1 =_\alpha e_2$ and $e_1 \rightarrow_\beta e_3$, then $e_2 \rightarrow_\beta e_4$ and $e_3 =_\alpha e_4$.



More terminology

$$(\lambda v.e_1) e_2 \rightarrow_{\beta} e_1[e_2/v]$$

- The left-hand-side $(\lambda v.e_1) e_2$ is called a *redex*.
- The right-hand-side $e_1[e_2/v]$ is called the *contractum*.



Rules, inductive definitions

An *inference rule* has a set of *assumptions* above a horizontal line, and a single *conclusion* below the line.

$$\frac{\textit{assum}_1 \quad \dots \quad \textit{assum}_n}{\textit{concl}} \quad \text{dummy rule}$$



Multi-step reduction

Redex reduction:

$$\overline{(\lambda v.e_1) e_2 \rightarrow_{\beta} e_1[e_2/v]} \text{ redex}$$

Beta-reduction can be applied in any context:

$$\frac{M \rightarrow_{\beta} M'}{M N \rightarrow_{\beta} M' N} \text{ app fun}$$

$$\frac{N \rightarrow_{\beta} N'}{M N \rightarrow_{\beta} M N'} \text{ app arg}$$

$$\frac{M \rightarrow_{\beta} M'}{\lambda v.M \rightarrow_{\beta} \lambda v.M'} \text{ fun}$$



Multi-step reduction

$$\frac{}{M \rightarrow_{\beta}^* M} \text{ ref}$$

$$\frac{M \rightarrow_{\beta}^* M' \quad M' \rightarrow_{\beta}^* M''}{M \rightarrow_{\beta}^* M''} \text{ trans}$$



Problems

- How is a program evaluated?
- What are the values (the result of a program evaluation)



Normal forms

- An expression is in *normal form* if it contains no redexes.
- Head normal form: $\lambda v_1 \dots \lambda v_n. y e_1 \dots e_m$
- Weak head normal form: $y e_1 \dots e_m$



Programs and normal forms

$$\begin{aligned} \circ &\equiv (\lambda x.x x) (\lambda y.y y) \\ Y &\equiv (\lambda f.(\lambda x.f(x x)) (\lambda x.f(x x))) \\ Y f &\rightarrow_{\beta}^* f (Y f) \end{aligned}$$

Y fact

$$\begin{aligned} &\equiv Y (\lambda f.\lambda i.\mathbf{if } i = 0 \mathbf{ then } 1 \mathbf{ else } i * (f(i - 1))) \\ &\rightarrow \lambda i.\mathbf{if } i = 0 \mathbf{ then } 1 \mathbf{ else } i * (Y \mathit{fact} (i - 1)) \end{aligned}$$



Normal forms

- It is probably best to stop as soon as a functional form is reached
- What about evaluation order?
- Does evaluation order matter?

$$\begin{aligned}(\lambda x.1) \text{ } \cdot &\rightarrow (\lambda x.1) \text{ } \cdot \\ &\rightarrow (\lambda x.1) \text{ } \cdot \\ &\rightarrow (\lambda x.1) \text{ } \cdot \\ &\rightarrow 1\end{aligned}$$



Church-Rosser

Church-Rosser: evaluation order doesn't matter.

- Theorem: if $M \rightarrow_{\beta}^* N_1$ and $M \rightarrow_{\beta}^* N_2$ then there is a term P such that $N_1 \rightarrow_{\beta}^* P$ and $N_2 \rightarrow_{\beta}^* P$.
- Corollary: if a term has a normal form, the normal form is unique.
- Proof: hard.



Natural semantics

- While beta-reduction is sufficient for reasoning about evaluation, it is not a very good method for automated evaluation
 - “Pick any redex in any subterm, reduce it, and continue”
- Natural semantics: define a *procedure* for program evaluation
- Also called “big-step” semantics (beta-reduction is “small-step” semantics)



Natural semantics

- Use v to represent a value (in weak head normal form)
- If $e \downarrow v$, we say “ e evaluates to the value v ”
- Evaluation rules:

$$\frac{}{x \downarrow x} \text{ var}$$

$$\frac{}{(\lambda x.e) \downarrow (\lambda x.e)} \text{ abs}$$



Natural semantics of application

- Two forms
 - *Eager*: evaluate the argument first
 - *Lazy*: do not evaluate the argument first

$$\frac{e_1 \downarrow (\lambda x.e_3) \quad e_2 \downarrow v_1 \quad e_3[v_1/x] \downarrow v_2}{e_1 \quad e_2 \downarrow v_2} \text{ eager}$$

$$\frac{e_1 \downarrow (\lambda x.e_3) \quad e_3[e_2/x] \downarrow v_2}{e_1 \quad e_2 \downarrow v_2} \text{ lazy}$$



Natural semantics

- Soundness:

Theorem If $e \downarrow v$ then $e \rightarrow_{\beta}^* v$.

Proof techniques:

1. Structural induction.
2. Proof induction.



Structural induction

Structural induction is a method for proving properties of all λ terms.

The induction principle is: assume that the induction principle holds for all smaller terms, and prove it for the term.

There are three cases to prove a property $P(e)$:

var Prove $P(x)$ (base case)

abs Prove $P(\lambda x.e)$ from $P(e)$

app Prove $P(e_1 e_2)$ from $P(e_1)$ and $P(e_2)$



Proof induction

Proof induction is induction on the length of a proof.
To prove a property $P(t)$ for an arbitrary judgment, for each rule

$$\frac{s_1 \quad \dots \quad s_n}{t}$$

prove $P(t)$ from $P(s_1), \dots, P(s_n)$.



Soundness of natural semantics

Theorem If $e \downarrow v$ then $e \rightarrow_{\beta}^* v$.

Proof For trivial cases,

- If $\overline{x \downarrow x}^{\text{var}}$, then $x \rightarrow_{\beta}^* x$.
- If $\overline{(\lambda x.e) \downarrow (\lambda x.e)}^{\text{abs}}$, then $(\lambda x.e) \rightarrow_{\beta}^* (\lambda x.e)$



Induction step

Otherwise, suppose the last rule was for an application:

$$\frac{e_1 \downarrow (\lambda x.e_3) \quad e_3[e_2/x] \downarrow v_2}{e_1 e_2 \downarrow v_2} \text{ lazy}$$

Chain these together:

	$e_1 e_2$	Initial program
\rightarrow_{β}^*	$(\lambda x.e_3) e_2$	by induction
\rightarrow_{β}^*	$e_3[e_2/x]$	beta reduction
\rightarrow_{β}^*	v_2	by induction

