Lambda Calculus

CS 152 (Spring 2020)

Harvard University

Thursday, February 20, 2020

Today, we will learn about

- Lambda calculus
- Full β-reduction
- Call-by-value semantics
- Call-by-name semantics

Lambda calculus: Intuition

A function is a rule for determining a value from an argument. Some examples of functions in mathematics are

$$f(x) = x^3$$
$$g(y) = y^3 - 2y^2 + 5y - 6.$$

Pure vs Applied Lambda Calculus

- The pure λ-calculus contains just function definitions (called *abstractions*), variables, and function *applications*.
- If we add additional data types and operations (such as integers and addition), we have an *applied* λ-calculus.

Pure Lambda Calculus: Syntax

$$\begin{aligned} e ::= \;& x && \text{variable} \\ \mid\;& \lambda x.\, e && \text{abstraction} \\ \mid\;& e_1 \; e_2 && \text{application} \end{aligned}$$

Abstractions

Abstractions

- An abstraction $\lambda x.\, e$ is a function
- Variable x is the *argument*
- Expression e is the *body* of the function.
- The expression $\lambda y.\, y \times y$ is a function that takes an argument y and returns square of y.

Applications

- An application $e_1\ e_2$ requires that e_1 is (or evaluates to) a function, and then applies the function to the expression e_2.

- For example, $(\lambda y.\, y \times y)\ 5$ is 25

Examples

$\lambda x.\, x$	a lambda abstraction called the *identity function*
$\lambda x.\, (f\ (g\ x)))$	another abstraction
$(\lambda x.\, x)\ 42$	an application
$\lambda y.\, \lambda x.\, x$	an abstraction, ignores its argument and returns the identity function

Lambda expressions extend as far to the right as possible

$\lambda x.\, x\ \lambda y.\, y$ is the same as $\lambda x.\, (x\ (\lambda y.\, y))$, and is not the same as $(\lambda x.\, x)\ (\lambda y.\, y)$.

Application is left-associative

$e_1\ e_2\ e_3$ is the same as $(e_1\ e_2)\ e_3$.

Use parentheses!

In general, use parentheses to make the parsing of a lambda expression clear if you are in doubt.

Variable binding

- An occurrence of a variable x in a term is bound if there is an enclosing $\lambda x.\, e$; otherwise, it is *free*.

- A *closed term* is one in which all identifiers are bound.

Variable binding: $\lambda x.\,(x\ (\lambda y.\, y\ a)\ x)\ y$

Variable binding: $\lambda x.\,(x\ (\lambda y.\,y\ a)\ x)\ y$

- Both occurrences of x are bound
- The first occurrence of y is bound
- The a is free
- The last y is also free, since it is outside the scope of the λy.

Binding operator

The symbol λ is a *binding operator*: variable x is bound in e in the expression $\lambda x.\, e$.

α-equivalence

- $\lambda x.\, x$ is the same function as $\lambda y.\, y$.
- Expressions e_1 and e_2 that differ only in the name of bound variables are called *α-equivalent* ("alpha equivalent")
- Sometimes written $e_1 =_\alpha e_2$.

Higher-order functions

- In lambda calculus, functions are values.
- In the pure lambda calculus, every value is a function, and every result is a function!

Higher-order functions

$$\lambda f.\, f\ 42$$

Higher-order functions

$$\lambda v.\, \lambda f.\, (f\ \ v)$$

Takes an argument v and returns a function that applies its own argument (a function) to v.

Semantics

β-equivalence

- We would like to regard $(\lambda x.\, e_1)\; e_2$ as equivalent to e_1 where every (free) occurrence of x is replaced with e_2.
- E.g. we would like to regard $(\lambda y.\, y \times y)\; 5$ as equivalent to 5×5.

$e_1\{e_2/x\}$

- We write $e_1\{e_2/x\}$ to mean expression e_1 with all free occurrences of x replaced with e_2.
- We call $(\lambda x.\, e_1)\ e_2$ and $e_1\{e_2/x\}$ β-*equivalent*.
- Rewriting $(\lambda x.\, e_1)\ e_2$ into $e_1\{e_2/x\}$ is called a β-*reduction*.
- This corresponds to executing a lambda calculus expression.

Different semantics for the lambda calculus

$$(\lambda x.\, x + x)\ ((\lambda y.\, y)\ 5)$$

Different semantics for the lambda calculus

$$(\lambda x.\, x + x)\ ((\lambda y.\, y)\ 5)$$

We could use β-reduction to get either $((\lambda y.\, y)\ 5) + ((\lambda y.\, y)\ 5)$ or $(\lambda x.\, x + x)\ 5$.

Evaluation strategies: Full β-reduction

Allows $(\lambda x.\, e_1)\ e_2$ to step to $e_1\{e_2/x\}$ at any time.

Full β-reduction: small-step operational semantics

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\ e_2 \longrightarrow e_1\ e_2'}$$

$$\frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}$$

$$\beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}}$$

Normal form

A term e is said to be in *normal form* when there is no e' such that $e \longrightarrow e'$.

Not every term has a normal form under full β-reduction.

Consider $\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$.

$$\Omega = (\lambda x.\, x\ x)\,(\lambda x.\, x\ x) \longrightarrow (\lambda x.\, x\ x)\,(\lambda x.\, x\ x) = \Omega$$

It's an infinite loop!

Well-behaved nondeterminism

$$(\lambda x.\, \lambda y.\, y)\ \Omega\ (\lambda z.\, z)$$

Well-behaved nondeterminism

$$(\lambda x.\, \lambda y.\, y)\ \Omega\ (\lambda z.\, z)$$

This term has two redexes in it, the one with abstraction λx, and the one inside Ω.

Well-behaved nondeterminism

- The full β-reduction strategy is non-deterministic.
- When a term has a normal form, however, it never has more than one.

Full β-reduction is confluent

Theorem (Confluence)

If $e \longrightarrow^* e_1$ *and* $e \longrightarrow^* e_2$ *then there exists* e' *such that* $e_1 \longrightarrow^* e'$ *and* $e_2 \longrightarrow^* e'$*.*

Full β-reduction is confluent

Corollary

If $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$ and both e_1 and e_2 are in normal form, then $e_1 = e_2$.

Proof.

An easy consequence of confluence. □

Normal Order Evaluation

- *Normal order evaluation* uses the full β-reduction rules, except the left-most redex is always reduced first.
- Will eventually yield the normal form, if one exists.
- Allows reducing redexes inside abstractions

Call-by-value

- *Call-by-value* only allows an application to reduce after its argument has been reduced to a value and does not allow evaluation under a λ.

- Given an application $(\lambda x.\, e_1)\ e_2$, CBV semantics makes sure that e_2 is a value before calling the function.

- A value is an expression that can not be reduced/executed/simplified any further.

CBV: Small step operational semantics

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

$$\frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}$$

$$\beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}$$

CBV: Examples

$$
\begin{aligned}
(\lambda x.\, \lambda y.\, y\ x)\ (5 + 2)\ \lambda x.\, x + 1 \quad &\longrightarrow (\lambda x.\, \lambda y.\, y\ x)\ 7\ \lambda x.\, x + 1 \\
&\longrightarrow (\lambda y.\, y\ 7)\ \lambda x.\, x + 1 \\
&\longrightarrow (\lambda x.\, x + 1)\ 7 \\
&\longrightarrow 7 + 1 \\
&\longrightarrow 8
\end{aligned}
$$

$$\begin{aligned}(\lambda f.\, f\ 7)\ ((\lambda x.\, x\ x)\ \lambda y.\, y) &\longrightarrow (\lambda f.\, f\ 7)\ ((\lambda y.\, y)\ (\lambda y.\, y)) \\ &\longrightarrow (\lambda f.\, f\ 7)\ (\lambda y.\, y) \\ &\longrightarrow (\lambda y.\, y)\ 7 \\ &\longrightarrow 7\end{aligned}$$

Call-by-name semantics

- More permissive that CBV.
- Less permissive than full β-reduction.
- Applies the function as soon as possible.
- No need to ensure that the expression to which a function is applied is a value.

Call-by-name semantics

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

$$\beta\text{-}\textsc{reduction}\ \frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}}$$

Call-by-name semantics: example

$$\begin{aligned}(\lambda x.\, \lambda y.\, y\ x)\ (5+2)\ \lambda x.\, x+1 \quad &\longrightarrow (\lambda y.\, y\ (5+2))\ \lambda x.\, x+1\\ &\longrightarrow (\lambda x.\, x+1)\ (5+2)\\ &\longrightarrow (5+2)+1\\ &\longrightarrow 7+1\\ &\longrightarrow 8\end{aligned}$$

compare to CBV:

$$\begin{aligned}(\lambda x.\, \lambda y.\, y\ x)\ (5+2)\ \lambda x.\, x+1 \quad &\longrightarrow (\lambda x.\, \lambda y.\, y\ x)\ 7\ \lambda x.\, x+1\\ &\longrightarrow (\lambda y.\, y\ 7)\ \lambda x.\, x+1\\ &\longrightarrow (\lambda x.\, x+1)\ 7\\ &\longrightarrow 7+1\\ &\longrightarrow 8\end{aligned}$$

Call-by-name semantics: example

$$\begin{aligned}(\lambda f.\, f\ 7)\ ((\lambda x.\, x\ x)\ \lambda y.\, y) &\longrightarrow ((\lambda x.\, x\ x)\ \lambda y.\, y)\ 7\\ &\longrightarrow ((\lambda y.\, y)\ (\lambda y.\, y))\ 7\\ &\longrightarrow (\lambda y.\, y)\ 7\\ &\longrightarrow 7\end{aligned}$$

compare to CBV:

$$\begin{aligned}(\lambda f.\, f\ 7)\ ((\lambda x.\, x\ x)\ \lambda y.\, y) &\longrightarrow (\lambda f.\, f\ 7)\ ((\lambda y.\, y)\ (\lambda y.\, y))\\ &\longrightarrow (\lambda f.\, f\ 7)\ (\lambda y.\, y)\\ &\longrightarrow (\lambda y.\, y)\ 7\\ &\longrightarrow 7\end{aligned}$$

CBV vs CBN

One way in which CBV and CBN differ is when arguments to functions have no normal forms.

$$(\lambda x.(\lambda y.y))\ \Omega$$

Under CBV semantics, this term does not have a normal form. If we use CBN semantics, then we have

$$(\lambda x.(\lambda y.y))\ \Omega \longrightarrow_{\mathsf{CBN}} \lambda y.y$$

CBV and CBN

- CBV and CBN are common evaluation orders
- Many programming languages use CBV semantics
- "Lazy" languages, such as Haskell, typically use CBN semantics, a more efficient semantics similar to CBN in that it does not evaluate actual arguments unless necessary
- However, Call-by-value semantics ensures that arguments are evaluated at most once.