The Lambda Calculus

CIS 352 — Spring 2020
The Lambda Calculus

• A system for calculating based entirely on computing with functions.

• Developed as a foundation for mathematics (originally used to model the natural numbers) by Alonzo Church in 1936.

• Church’s thesis: “Every effectively calculable function (effectively decidable predicate) is general recursive”, i.e., can be computed using the λ-calculus. Used to show there exist unsolvable problems.

• One of the simplest Turing-equivalent languages!

  • Church, with his student Alan Turing, proved the equivalent expressiveness of Turing machines and the λ-calculus (called the Church-Turing thesis).

• Still makes up the heart of all functional programming languages!
The Lambda Calculus

*lamdas* are just anonymous functions!

\[
e \in \text{Exp} ::= (\lambda (x) e) \quad \text{\(\lambda\)-abstraction}
\]

| \(e \ e\) \quad \text{function application}
| \(x\) \quad \text{variable reference}

\[
x \in \text{Var} ::= \langle\text{variables}\rangle
\]
Textual-reduction semantics

• One way of designing a formal semantics is as a relation over terms in the language—one that reduces the term textually.

• This is usually small-step—each eval step must terminate (meaning there are no premises above the line in our rules of inference and no recursive use of the interpreter within a step.)

• Consider a small-step semantics for our arithmetic language:

\[ a \in \text{AExp} ::= n \mid a + a \mid a - a \mid a \times a \]

\[ n, m \in \text{Num} ::= \langle \text{integer constants} \rangle \]
Exercise

Which of the following are AExps:

• 10
• 20.5
• 10 + 3
• 10 + 3 * 4²
• 5 - 3 * 2 + 3 * 1

\[ a \in \text{AExp} ::= n \mid a + a \mid a - a \mid a \times a \]

\[ n, m \in \text{Num} ::= \langle \text{integer constants} \rangle \]
Exercise

Which of the following are AExp:

• 10
• 20.5 — No, not integer constant
• 10 + 3
• 10 + 3 * 4^2 — Exponent not allowed
• 5 - 3 * 2 + 3 * 1

\[
a \in \text{AExp} \quad ::= \quad n \mid a + a \mid a - a \mid a \times a
\]

\[
n, m \in \text{Num} \quad ::= \quad \langle \text{integer constants} \rangle
\]
Textual-reduction semantics

\[ a \in AExp ::= n \mid a + a \mid a - a \mid a \times a \]

\[ n, m \in \text{Num} ::= \langle \text{integer constants} \rangle \]

- Rules to reduce terms in this language match operations that have two numeric operands already and apply the operation, textually substituting a numeric value for the operation; e.g.:

\[
a_0 \times a_1 \Rightarrow n_0 \times n_1
\]

where \( a_0 \) is \( n_0 \) and \( a_1 \) is \( n_1 \)

- For example:

\[
2 \times 3 + 4 \times 5 \Rightarrow 2 \times 3 + 20 \Rightarrow 6 + 20 \Rightarrow 26
\]

- Is there another way to evaluate \( 2 \times 3 + 4 \times 5 \) using similar rules?
The Lambda Calculus

*lamdas* are just anonymous functions!

\[
e \in \text{Exp} ::= (\lambda \ (x) \ e) \quad \text{\(\lambda\)-abstraction}
\]
\[
| \ (e \ e) \quad \text{function application}
\]
\[
| \ x \quad \text{variable reference}
\]

\[
x \in \text{Var} ::= \langle \text{variables} \rangle
\]
The lambda-calculus is the functional core of Racket (as of other functional languages).

Just the following subset of Racket is Turing-equivalent!

\[ e \in \text{Exp} ::= (\lambda (x) e) \quad (\text{lambda} \ (x) \ e) \]
\[ \quad | \ (e \ e) \quad (e_0 \ e_1) \]
\[ \quad | \ x \quad x \]

\[ x \in \text{Var} ::= \langle \text{variables} \rangle \]
Lambda Abstraction

An expression, *abstracted* over all possible values for a formal parameter, in this case, $x$.

$$(\lambda (x) e)$$

- Formal parameter
- Function body
Application

When the first expression is evaluated to a value (in this language, all values are functions!) it may be invoked / applied on its argument.

\[(e \ e)\]

Expression in \textit{function position}

Expression in \textit{argument position}
Variables

Variables are only defined/assigned when a function is applied and its parameter bound to an argument.
We define a rule for step-by-step evaluation called **Beta-reduction**.
Textual substitution. This says: *replace every $x$ in $E_0$ with $E_1$.*

$$(((\lambda (x) \ E_0) \ E_1)) \rightarrow_\beta \ E_0 [x \leftarrow E_1]$$

*redex*

*(reducible expression)*
$$(((\lambda(x)x)(\lambda(x)x))$$

\[\xrightarrow{\beta}\]

$$x[x \leftarrow (\lambda(x)x)]$$
\[(\lambda (x) x) (\lambda (x) x)\]
Can you beta-reduce the following term more than once:

\[ (((\lambda x)(x x))(\lambda x)(x x)) \]
\[
(((\lambda (x) (x x)) \ (\lambda (x) (x x))))
\]

β reduction may continue indefinitely (i.e., in non-terminating programs)

\[
(((\lambda (x) (x x)) \ (\lambda (x) (x x)))) \rightarrow \beta
\]

\[
(((\lambda (x) (x x)) \ (\lambda (x) (x x)))) \rightarrow \beta
\]

\[
(((\lambda (x) (x x)) \ (\lambda (x) (x x)))) \rightarrow \beta
\]

\[
(((\lambda (x) (x x)) \ (\lambda (x) (x x)))) \rightarrow \beta
\]
This specific program is known as \( \Omega \) (Omega)
\( \Omega \) is the smallest non-terminating program!

Note how it reduces to itself in a single step!
Example

Evaluation with $\beta$ reduction is nondeterministic!

$$(((\lambda (w) w) (\lambda (x) x)) \ ((\lambda (y) y) \ (\lambda (z) z)))$$

$$\xrightarrow{\beta}$$

$$(((\lambda (x) x) \ ((\lambda (y) y) \ (\lambda (z) z)))$$
Example

Evaluation with $\beta$ reduction is nondeterministic!

$$(((\lambda (w) w) (\lambda (x) x)) (\lambda (w) w) (\lambda (x) x))$$

$\beta$ or!

$$(((\lambda (x) x) (\lambda (y) y) (\lambda (z) z))$$

$\beta$

$$(((\lambda (x) x) (\lambda (y) y) (\lambda (z) z)))$$

$\beta$

$$(((\lambda (w) w) (\lambda (x) x)) (\lambda (z) z))$$

22
Exercise

Perform each possible β-reduction

\(((\lambda \ (x)) \ (\ (\lambda \ (y) \ (x \ y)) \ x)) \ (\lambda \ (z) \ (z \ z)))\)

How many different β-reductions are possible from the above?
Can reduce inner redex...
Exercise

\[ (((\lambda \, x) \, (((\lambda \, y) \, (x \, y)) \, x)) \, (\lambda \, (z) \, (z \, z))) \]

\[ \beta \]

\[ (((\lambda \, (y)) \, (((\lambda \, (z)) \, (z \, z)) \, y)) \, (\lambda \, (z) \, (z \, z))) \]

Or the outer redex.
Can’t reduce this since we don’t (yet) know about the particular value (function) \( z \) in call position.
Free Variables

We define the free variables of a lambda expression via the function $FV$:

$$FV : \text{Exp} \rightarrow \mathcal{P}(\text{Var})$$

$$FV(x) \triangleq \{x\}$$

$$FV((\lambda (x) \ e_b)) \triangleq FV(e_b) \setminus \{x\}$$

$$FV(e_f \ e_a) \triangleq FV(e_f) \cup FV(e_a)$$
Example

\[
\text{FV}((x \ y)) = \{x, y\}
\]

\[
\text{FV}(((\lambda (x) x) \ y)) = \{y\}
\]

\[
\text{FV}(((\lambda (x) x) \ x)) = \{x\}
\]

\[
\text{FV}(((\lambda (y) ((\lambda (x) (z \ x)) \ x))) = \{z, x\}
\]
What are the free variables of each of the following terms?

\[
((\lambda (x) \; x) \; y)
\]

\[
((\lambda (x) \; (x \; x)) \; (\lambda (x) \; (x \; x)))
\]

\[
((\lambda (x) \; (z \; y)) \; x)
\]
What are the free variables of each of the following terms?

\[
((\lambda (x) x) y)
\]

\{y\}

\[
(((\lambda (x) (x x)) (\lambda (x) (x x)))
\]

\{\}

\[
(((\lambda (x) (z y)) x)
\]

\{x, y, z\}
The problem with (naive) textual substitution

\[
((\lambda (a) (\lambda (a) a)) \ (\lambda (b) b)) \ \beta
\]
The problem with (naive) textual substitution

\[
((\lambda (a) (\lambda (a) a)) (\lambda (b) b))
\]

\[\beta\]

\[
(\lambda (a) a)[a \leftarrow (\lambda (b) b)]
\]
The problem with (naive) textual substitution

\[
((\lambda (a) (\lambda (a) a)) (\lambda (b) b)) \quad \xrightarrow{\beta} \quad (\lambda (a) (\lambda (b) b))
\]
Capture-avoiding substitution

\[ E_\theta[x \leftarrow E_1] \]
\[
x[x \leftarrow E] = E
\]

\[
y[x \leftarrow E] = y \text{ where } y \neq x
\]

\[
(E_0 \ E_1)[x \leftarrow E] = (E_0[x \leftarrow E] \ E_1[x \leftarrow E])
\]

\[
(\lambda (x) \ E_0)[x \leftarrow E] = (\lambda (x) \ E_0)
\]

\[
(\lambda (y) \ E_0)[x \leftarrow E] = (\lambda (y) \ E_0[x \leftarrow E])
\]

where \( y \neq x \) and \( y \not\in \text{FV}(E) \)

\( \beta \)-reduction cannot occur when \( y \in \text{FV}(E) \)
Capture-avoiding substitution

\[ (((\lambda (a)) (\lambda (a) a)) (\lambda (b) b)) \]

\[ \beta \]

\[ (\lambda (a) a) \]
Exercise

How can you beta-reduce the following expression using capture-avoiding substitution?

\(((\lambda \ (y)) \ ((\lambda \ (z) \ (\lambda \ (y) \ (z \ y))) \ y)) \ (\lambda \ (x) \ x))\)
Exercise

How can you beta-reduce the following expression using capture-avoiding substitution?

$$(((\lambda (y) (((\lambda (z) (\lambda (y) (z \ y))) y) y)) (\lambda (x) x)) \xrightarrow{\beta} (((\lambda (z) (\lambda (y) (z \ y))) (\lambda (x) x))$$
How can you beta-reduce the following expression using capture-avoiding substitution?

\[(\lambda \ y \ (\ (\lambda \ z \ (\lambda \ y \ z) \ ) \ (\lambda \ x \ y) \ ))\]
Exercise

How can you beta-reduce the following expression using capture-avoiding substitution?

$$(\lambda (y) \ ((\lambda (z) \ (\lambda (y) \ z)) \ (\lambda (x) \ y)))$$

You cannot! This redex would require:

$$(\lambda (y) \ z)[z \leftarrow (\lambda (x) \ y)]$$

(y is free here, so it would be captured)
How can you beta-reduce the following expression using capture-avoiding substitution?

\[(\lambda (y) \ ((\lambda (z) \ (\lambda (y) \ z)) \ (\lambda (x) \ y))))\]

\[\rightarrow_\alpha (\lambda (y) \ ((\lambda (z) \ (\lambda (w) \ z)) \ (\lambda (x) \ y))))\]

\[\rightarrow_\beta (\lambda (y) \ (\lambda (w) \ (\lambda (x) \ y))))\]

Instead we alpha-convert first.
\( \alpha \)-renaming

\[
(\lambda (x) (\lambda (y) x)) \quad (\lambda (a) (\lambda (b) a))
\]

These two expressions are equivalent—they only differ by their variable names \((x = a; y = b)\)
\( \alpha \)-renaming

\[
(\lambda (x) \ E_\theta) \quad \rightarrow_{\alpha} \quad (\lambda (y) \ E_\theta[x \leftarrow y])
\]

\[=\alpha \]

\(\alpha\) renaming/conversions can be run backward, so you might think of it as an equivalence relation.
α-renaming

α-renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

\[
((\lambda (x) (\lambda (x) x)) (\lambda (y) y))
\]
\( \alpha \)-renaming

\( \alpha \) renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

\[
((\lambda (x) (\lambda (x) x)) (\lambda (y) y))
\]

Can’t perform naive substitution w/o capturing x.
\( \alpha \)-renaming

\( \alpha \) renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

\[
((\lambda (x) (\lambda (x) x)) (\lambda (y) y))
\]

Fix by \( \alpha \) renaming to z
\( \alpha \)-renaming

\( \alpha \) renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

\[
((\lambda (x) (\lambda (z) z)) (\lambda (y) y))
\]

Fix by \( \alpha \) renaming to \( z \)
α-renaming

α-renaming/conversions can be used to implement capture-avoiding substitution

Rename variables that would break naive substitution!

\[
(((\lambda (x) (\lambda (z) z)) (\lambda (y) y))
\]

Could now perform beta-reduction with naive substitution
\( \eta \)-reduction

\[
(\lambda (x) (E_\theta x)) \xrightarrow{\eta} E_\theta \quad \text{where } x \notin \text{FV}(E_\theta)
\]
$\eta$-expansion

\[ E_0 \rightarrow_\eta (\lambda (x) (E_0 \ x)) \quad \text{where} \ x \notin \text{FV}(E_0) \]
Reduction

\[(\rightarrow) = (\rightarrow_\beta) \cup (\rightarrow_\alpha) \cup (\rightarrow_\eta)\]

\[(\rightarrow^*)\]

reflexive/transitive closure
Evaluation

\[ E_0 \]
\[ \rightarrow \]
\[ * \]
\[ \rightarrow \]
\[ E_8 \]
\[ \rightarrow \]
\[ * \]
\[ \rightarrow \]
\[ ? \]
Evaluation to *normal form*

\[ E_0 \]

\[ \ast \]

\[ (\lambda \ (x) \ ... ) \]
Evaluation to *normal form*

\[ E_0 \]

\[ * \]

\[ (\lambda (x) \ldots (\lambda (z) ((a \ldots) \ldots))) \]

In *normal form*, no function position can be a lambda; this is to say: *there are no unreduced redexes left!*
Evaluation Strategy

E₀

*   *

E₁   E₂
Evaluation Strategy

\[ (((\lambda (x) ((\lambda (y) y) x)) (\lambda (z) z)) \rightarrow_{\eta} ((\lambda (y) y) (\lambda (z) z)) \rightarrow_{\beta} (\lambda (z) z)) \]
Evaluation Strategy

\[( ((\lambda \ (x)) \ ((\lambda \ (y) \ y) \ x)) \ (\lambda \ (z) \ z)) \]

\[\rightarrow_{\beta} \ ((\lambda \ (y) \ y) \ (\lambda \ (z) \ z)) \]

\[\rightarrow_{\beta} \ (\lambda \ (z) \ z) \]
Evaluation Strategy

\[
\left( (\lambda \ x) \left( (\lambda \ (y) \ y) \ x \right) \right) \ (\lambda \ (z) \ z)
\]

\[\rightarrow_\beta \left( (\lambda \ (x) \ x) \ (\lambda \ (z) \ z) \right) \]

\[\rightarrow_\beta (\lambda \ (z) \ z)\]
Confluence

Diverging paths of evaluation must eventually join back together.

Church-Rosser Theorem
\[ e_0 \xleftarrow{\alpha} e_1 \xleftarrow{\alpha} e_2 \xleftarrow{\alpha} e_3 \xleftarrow{\alpha} e_4 \xleftarrow{\alpha} e_5 \]
Confluence (i.e., Church-Rosser Theorem)
Applicative evaluation order

Always evaluates the *innermost* leftmost redex first.

Normal evaluation order

Always evaluates the *outermost* leftmost redex first.
Applicative evaluation order

\[ (((\lambda(x) \ (\lambda(y) \ y) \ x)) \ (\lambda(z) \ z)) \]

Normal evaluation order

\[ (((\lambda(x) \ (\lambda(y) \ y) \ x)) \ (\lambda(z) \ z)) \ (\lambda(w) \ w)) \]
Call-by-value (CBV) semantics

Applicative evaluation order, but not under lambdas.

Call-by-name (CBN) semantics

Normal evaluation order, but not under lambdas.
Exercise

Write a lambda term other than $\Omega$ which also does not terminate

(Hint: think about using some form of self-application)
Exercise

Write a lambda term other than Ω which also does not terminate

\[
((\lambda \ (y) \ ((\lambda \ (x) \ (y \ x)) \ y)) \\
(\lambda \ (y) \ ((\lambda \ (x) \ (y \ x)) \ y)))
\]

\[
((\lambda \ (u) \ ((u \ u) \ u)) \\
(\lambda \ (u) \ ((u \ u) \ u)))
\]

\[
((\lambda \ (x) \ x) \\
((\lambda \ (u) \ (u \ u)) \\
(\lambda \ (u) \ (u \ u))))
\]