CSc 520
Principles of Programming Languages
22: Lambda Calculus — Reductions

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg
Lambda Reductions

To evaluate a lambda expression we reduce it until we can apply no more reduction rules. There are four principal reductions that we use:

1. $\alpha$-reduction — variable renaming to avoid name clashes in $\beta$-reductions.
2. $\beta$-reduction — function application.
3. $\eta$-reduction — formula simplification.
4. $\delta$-reduction — evaluation of predefined constants and functions.
Blindly applying $\beta$-reductions can lead to a problem known as **variable capture**. To prevent this we need a way to change the name of a variable.

Here’s an example of an $\alpha$-reduction:

$$\Rightarrow \alpha$$

$$(\lambda z.(\lambda f.(f \ x))) \ z$$
The equivalence of function application in a functional language is called \( \beta \)-reduction.

Here’s an example of using \( \beta \)-reductions to evaluate a lambda expression:

\[
\text{Twice} \equiv (\lambda f.(\lambda x.(f \ (f \ x))))
\]

\[
(((\text{Twice} \ (\lambda n.(\text{add} \ n \ 1)))) \ 5) \Rightarrow \beta
\]

\[
(((\lambda f.(\lambda x.(f \ (f \ x)))) \ (\lambda n.(\text{add} \ n \ 1))) \ 5) \Rightarrow \beta
\]

\[
((\lambda x.((\lambda n.(\text{add} \ n \ 1)) \ ((\lambda n.(\text{add} \ n \ 1)) \ x))) \ 5) \Rightarrow \beta
\]

\[
((\lambda n.(\text{add} \ n \ 1)) \ ((\lambda n.(\text{add} \ n \ 1)) \ 5)) \Rightarrow \beta
\]
\( \beta \)-reduction — Introductory Example...

\[
((\text{Twice } (\lambda n. (\text{add } n \ 1))) \ 5) \Rightarrow \\
(((\lambda f. (\lambda x. (f \ (f \ x)))) \ (\lambda n. (\text{add } n \ 1))) \ 5) \Rightarrow_\beta \\
((\lambda x. ((\lambda n. (\text{add } n \ 1)) \ ((\lambda n. (\text{add } n \ 1)) \ x))) \ 5) \Rightarrow_\beta \\
((\lambda n. (\text{add } n \ 1)) \ ((\lambda n. (\text{add } n \ 1)) \ 5)) \Rightarrow_\beta \\
(\text{add } ((\lambda n. (\text{add } n \ 1)) \ 5) \ 1) \Rightarrow_\beta \\
(\text{add } (\text{add } 5 \ 1) \ 1)
\]

\( \bullet \) No further \( \beta \)-reductions can be made.
A $\delta$-reduction is used to evaluate non-pure lambda expression, i.e. those that contain predefined constants and functions (such as $\text{add}$, etc.).

Without $\delta$-reductions we couldn’t evaluate the previous example any further:

$$(((\text{Twice} (\lambda n. (\text{add} \ n \ 1))) \ 5) \Rightarrow *)$$

$$(\text{add} \ (\text{add} \ 5 \ 1) \ 1) \Rightarrow \delta$$

$$(\text{add} \ 6 \ 1) \Rightarrow \delta$$

7
Free Variables
Free Variable Substitution

To work with $\alpha$-reductions we first need to define the concept of a variable substitution.

The notation

$$E[v \rightarrow M]$$

means “replace all free occurrences of $v$ with $M$ in the expression $E$.”

Example:

$$\left(\lambda v. (\text{mul } y v)\right)[y \rightarrow z] = (\lambda v. (\text{mul } z v))$$
A more traditional notation (used by Scott, for example) for
\[ E[v \rightarrow M] \]
is
\[ E[M \setminus v] \]

Example:
\[ (\lambda v.(\text{mul } y \ v))[z \setminus y] = (\lambda v.(\text{mul } z \ v)) \]
Free Variable Substitution

- Variable replacements aren’t legal in

\[ E[v \rightarrow M] \]

if a free variable in M becomes bound.

- For example, the substitution

\( (\lambda x. (\text{mul } y x))[y \rightarrow x] \Rightarrow (\lambda x. (\text{mul } x x)) \)

is illegal because it causes a change in semantics: the new expression represents a squaring operation, the original doesn’t.

- This is known as variable capture or a name clash.
Computing Free Variables

The free variables of an expression $E$, $\text{FV}(E)$, is computed as

1. $\text{FV}(c) = \emptyset$ for any constant $c$.
2. $\text{FV}(x) = \{x\}$ for any variable $x$.
3. $\text{FV}((E_1 E_2)) = \text{FV}(E_1) \cup \text{FV}(E_2)$.
4. $\text{FV}((\lambda x.E)) = \text{FV}(E) - \{x\}$.

An expression that has no free variables ($\text{FV}(E) = \emptyset$) is called closed.
Computing Free Variables — Example

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

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

\[ (\text{FV}(y) \cup \text{FV}((\lambda y.((y \ x) \ z)))) - \{x\} = \]

\[ (\{y\} \cup (\text{FV}(((y \ x) \ z)) - \{y\})) - \{x\} = \]

\[ (\{y\} \cup ((\text{FV}(y) \cup \text{FV}(x)) \cup \{z\}) - \{y\}) - \{x\} = \]

\[ (\{y\} \cup (((\{y\} \cup \{x\}) \cup \{z\}) - \{y\}) - \{x\} = \]

\[ \{y, z\} \]
Variable Substitution — Algorithm

a) \( v[v \rightarrow E_1] = E_1 \) for any variable \( v \)

b) \( x[v \rightarrow E_1] = x \) for any variable \( x \neq v \)

c) \( c[v \rightarrow E_1] = c \) for any constant \( c \)

d) \((E_1 \ E_2)[v \rightarrow E_3] = (E_1[v \rightarrow E_3] \ E_2[v \rightarrow E_3])\)

e) \((\lambda v. E)[v \rightarrow E_1] = (\lambda v. E)\)

f) \((\lambda x. E)[v \rightarrow E_1] = (\lambda x. E[v \rightarrow E_1]), \text{ when } x \neq v \text{ and } x \not\in \text{FV}(E_1)\)

g) \((\lambda x. E)[v \rightarrow E_1] = (\lambda z. E[x \rightarrow z][v \rightarrow E_1]), \text{ when } x \neq v \text{ and } x \in \text{FV}(E_1) \text{ where } z \neq v \text{ and } z \not\in \text{FV}((E \ E_1))\)

In g) the first substitution \( E[x \rightarrow z] \) replaces a bound variable \( x \) by a new bound variable \( z \).
Variable Substitution — Example

\[(\lambda y.((\lambda f. (f \ x)) \ y))[x \rightarrow (f \ y)] \xrightarrow{g)\}

\[(\lambda z.((\lambda f. (f \ x)) \ y)[y \rightarrow z][x \rightarrow (f \ y)]) =

\[(\lambda z.((\lambda f. (f \ x)) \ z)[x \rightarrow (f \ y)]) \xrightarrow{d)\}

\[(\lambda z.((\lambda f. (f \ x))[x \rightarrow (f \ y)] \ z[x \rightarrow (f \ y)]) \xrightarrow{b)\}

\[(\lambda z.((\lambda g. (g \ x))[x \rightarrow (f \ y)] \ z)) \xrightarrow{d,b,a)}

\[(\lambda z.((\lambda g. (g \ (f \ y))) \ z))\]
Reductions
The main rule for evaluating a lambda expression is called $\beta$-reduction.

$\beta$-reduction is similar to function application in a functional language.

To prevent variable capture (when a free variable in $E_1$ becomes bound during the substitution $E[v \rightarrow E_1]$), we also need $\alpha$-reductions.

$\delta$-reductions are used to evaluate predefined functions such as $\text{add}$.

$\eta$-reductions are not strictly necessary but can be used to clean up messy expressions.
\(\alpha\)-reductions

Scott calls this \(\alpha\)-conversion.

\(\alpha\)-reduction says that we can replace \(v\) by \(w\) in \((\lambda v. E)\) as long as \(w\) does not occur free in \(E\).

Formally,

\[
(\lambda v. E) \Rightarrow_\alpha (\lambda w. E[v \rightarrow w])
\]

where \(w\) does not occur free in \(E\).

Example:

\[
(\lambda y. ((\lambda f. (f \ x)) \ y)) \Rightarrow_\alpha (\lambda z. ((\lambda f. (f \ x)) \ z))
\]

Example:

\[
(\lambda z. ((\lambda f. (f \ x)) \ z)) \Rightarrow_\alpha (\lambda z. ((\lambda g. (g \ x)) \ z))
\]
Let \( v \) be a variable and \( E \) and \( E_1 \) lambda expressions, then

\[
((\lambda v. E) \ E_1) \Rightarrow \beta \ E[v \rightarrow E_1]
\]

provided \( E[v \rightarrow E_1] \) is carried out safely.

The intuition is that the argument \( E_1 \) is passed to the function \((\lambda v. E)\) by substituting \( E_1 \) for the formal parameter \( v \).

An expression of the form

\[
((\lambda v. E) \ E_1)
\]

is called a \( \beta \)-redex (reduction expression), i.e. an expression that can be \( \beta \)-reduced.
\(\eta\)-reductions allow us to get rid of extra lambda abstractions.

For example, we can say

\[(\lambda x. (\text{square } x)) \Rightarrow_\eta \text{ square}\]

square is a function that squares its argument, while

\[(\lambda x. (\text{square } x))\] is a function of \(x\) that squares \(x\).

\[(\lambda x. (\text{square } x))\] reminds us that \text{square} takes an argument \(x\), but \text{square} is a bit less messy.
\( \eta \)-reductions...

- If \( E \) is a lambda expression that denotes a function, and \( \nu \) has no free occurrences in \( E \), then

\[
(\lambda \nu.(E \, \nu)) \Rightarrow_{\eta} E
\]

- The following reduction

\[
(\lambda x.(\text{add} \, x \, x)) \Rightarrow_{\eta} (\text{add} \, x)
\]

is invalid since \( (\text{add} \, x) \) has a free occurrence of \( x \).
Reduction Strategies
Termination

Question:
Can every lambda expression be reduced to a normal form?

Consider the expression

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

which reduces to itself:

\[(\lambda x.(x \ x)) (\lambda x.(x \ x)) \Rightarrow \beta\]

\[(\lambda x.(x \ x)) (\lambda x.(x \ x)) \Rightarrow \beta\]

\[\cdots\]

Answer: No.
Lambda calculus contains non-terminating reductions.
Question:
Is there more than one way to reduce a lambda expression?

Consider the expression

\[((\lambda x.(\lambda y.(\text{add } y ((\lambda z.(\text{mul } x z)) 3))) 7) 5)\]

Let’s try reducing it in two different ways, to see if we’ll arrive at different normal forms.

A lambda expression is in a normal form if we can apply no more $\beta$-reductions or $\delta$-reductions.
Paths...

$$(((\lambda x.(\lambda y.(\text{add } y ((\lambda z.(\text{mul } x z)) 3)))) 7) 5) =$$

$$(((\lambda x.(\lambda y.(\text{add } y ((\lambda z.(\text{mul } x z)) 3)))) 7) 5) \Rightarrow_\beta$$

$$((\lambda y.(\text{add } y ((\lambda z.(\text{mul } 7 z)) 3))) 5) \Rightarrow_\beta$$

$$(\text{add } 5 ((\lambda z.(\text{mul } 7 z)) 3)) \Rightarrow_\beta$$

$$(\text{add } 5 (\text{mul } 7 3)) \Rightarrow_\delta$$

$$(\text{add } 5 21) \Rightarrow_\delta$$

26
Paths...

\[((\lambda x. (\lambda y. (add y ((\lambda z. (mul x z)) 3))) 7) 5) =

(((\lambda x. (\lambda y. (add y ((\lambda z. (mul x z)) 3))) 7) 5) \Rightarrow_\beta

(((\lambda x. (\lambda y. (add y (mul x 3))) 7) 5) \Rightarrow_\beta

((\lambda y. (add y (mul 7 3))) 5) \Rightarrow_\delta

((\lambda y. (add y 21)) 5) \Rightarrow_\delta

(add 5 21) \Rightarrow_\delta

26
Question:

Is there more than one way to reduce a lambda expression?

Answer: Yes.
Consider the expression

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

We can either reduce the leftmost redex first:

$$(((\lambda y.5) ((\lambda x. (x x)) (\lambda x. (x x)))) \Rightarrow_\beta 5$$

or, we can evaluate the rightmost redex every time:

$$(((\lambda y.5) ((\lambda x. (x x)) (\lambda x. (x x)))) \Rightarrow_\beta$$

$$(((\lambda y.5) ((\lambda x. (x x)) (\lambda x. (x x)))) \Rightarrow_\beta$$

$$(((\lambda y.5) ((\lambda x. (x x)) (\lambda x. (x x)))) \Rightarrow_\beta \cdots$$
The **leftmost redex** is that redex whose $\lambda$ is textually to the left of all other redexes within the expression.

An **outermost redex** is defined to be a redex which is not contained within any other redex.

An **innermost redex** is defined to be a redex which contains no other redex.

A **normal order** reduction always reduces the leftmost outermost $\beta$-redex (or $\delta$-redex) first.

A **applicative order** reduction always reduces the leftmost innermost $\beta$-redex (or $\delta$-redex) first.
Finding Redexes

Remembering that a redex is an expression of the form \((\lambda x.E) \, y\), find the redexes in this expression:

\[
(((\lambda x.(\lambda y.(add \, x \, y)))) \, ((\lambda z.(succ \, z)) \, 5)) \, ((\lambda w.(sqr \, w)) \, 7)) = \\
(((\lambda x.(\lambda y.(add \, x \, y)))) \, (\lambda z.(succ \, z)) \, 5)) \, ((\lambda w.(sqr \, w)) \, 7)) = \\
(((\lambda x.(\lambda y.(add \, x \, y)))) \, (\lambda z.(succ \, z)) \, 5)) \, (\lambda w.(sqr \, w)) \, 7)) = \\
((\lambda x.(\lambda y.(add \, x \, y)))) \, (\lambda z.(succ \, z)) \, 5)) \, (\lambda w.(sqr \, w)) \, 7))
\]
Finding Redexes...

- If we have problems finding the redexes of an expression, we can first draw it as an abstract syntax tree.

This is a tree that shows the structure of an expression, ignoring syntactic details.

\[
\begin{align*}
\lambda x.y & \quad \text{lamb} \\
\text{comb} & \\
& \text{lamb} \\
& \text{comb} \\
& \text{lamb} \\
& \text{comb} \\
& \text{comb} \\
& \text{comb} \\
& \text{comb}
\end{align*}
\]

\[
\begin{align*}
(x \ (y \ z)) & \\
((z \ y) \ x) & \\
(x \ (\lambda y.z)) & \\
(\lambda x.(y \ z)) & \\
(x \ y)
\end{align*}
\]

- comb stands for an application, lamb for an abstraction.
Finding Redexes...

- During evaluation we look for a redex, a tree with the following structure:

```
  comb
     /
    lamb
       /
      x
       E
```

```
((\x.E) y)
```

- I.e., a redex is an abstraction joined with another expression to be passed to the function.
Finding Redexes...

- Again, consider the expression:

\[
\left(\left(\lambda x. (\lambda y. (\text{add } x \; y))\right) \; \left(\lambda z. (\text{succ } z)\right) \; 5\right) \; \left(\lambda w. (\text{sqr } w)\right) \; 7
\]

- Its abstract syntax tree is given on the next slide.

- The leftmost outermost redex is

\[
\left(\left(\lambda x. (\lambda y. (\text{add } x \; y))\right) \; \left(\lambda z. (\text{succ } z)\right) \; 5\right) \; \left(\lambda w. (\text{sqr } w)\right) \; 7
\]

- The leftmost innermost redex is

\[
\left(\left(\lambda x. (\lambda y. (\text{add } x \; y))\right) \; \left(\lambda z. (\text{succ } z)\right) \; 5\right) \; \left(\lambda w. (\text{sqr } w)\right) \; 7
\]
Finding Redexes...
Church-Rosser Theorem I

**Question:**
If there is more than one reduction strategy, does each one lead to the same normal form expression?

**Theorem:**
For any lambda expressions $E$, $F$ and $G$, if $E \Rightarrow^* F$ and $E \Rightarrow^* G$, there is a lambda expression $Z$ such that $F \Rightarrow^* Z$ and $G \Rightarrow^* Z$.

- diamond property
- confluence property
Corollary:

For any lambda expressions $E$, $M$ and $N$, if $E \overset{*}{\Rightarrow} M$ and $E \overset{*}{\Rightarrow} N$, where $M$ and $N$ are in normal form, $M$ and $N$ are variants of each other (except for changes in variables, using $\alpha$-reductions).

Answer: Yes, if a lambda expression is in normal form, it is unique, except for changes in bound variables.
Church-Rosser Theorem II

Question:
Is there a reduction strategy that will guarantee that a normal form expression will be produced, if one exists?

Theorem:
For any lambda expressions $E$ and $N$, if $E \overset{*}{\Rightarrow} N$ where $N$ is in normal form, there is a normal order reduction from $E$ to $N$.

Answer: Yes, normal order reduction will produce a normal form lambda expression, if one exists.
Normal Order Reduction

A normal-order reduction can have these outcomes:
1. A unique normal form lambda expression is reached (up to $\alpha$-reduction).
2. The reduction never terminates.
Church’s Theses

The effectively computable functions on the positive integers are precisely those functions definable in the pure lambda calculus (and computable by Turing Machines).

- It can be shown that every Turing Machine can be simulated by a lambda expression.
- It can be shown that every lambda expression can be realized by a Turing Machine.
- Hence, Turing Machines and lambda calculus are equivalent.
- Since it’s not possible to determine whether a Turing Machine will terminate, it’s not possible to determine whether a normal order reduction will terminate.
People
Born on June 14 (Flag Day), 1903, in Washington, D.C.

His great-grandfather (also named Alonzo Church) was a professor of mathematics and astronomy.

His grandfather Alonzo Webster Church was at one time Librarian of the U.S. Senate.

An airgun incident in high school left Church blind in one eye.

Church enrolled at Princeton, where his uncles had attended college. He worked part-time in the dining hall to help pay his way.
He was an exceptional student; his first published paper was written while he was an undergraduate.

He continued graduate work at Princeton, completing a Ph.D. in three years.

While a graduate student, he married Mary Julia Kuczinski, who was training to be a nurse. (This in spite of the fact that his senior class had voted him the most “likely to remain a bachelor”.

In the summer of 1924, Church stepped off the curb and was hit by a trolley car coming from his blind side; Mary was a nurse-trainee at the hospital.

Mary was an excellent cook; over the years many a mathematician enjoyed dining at the Church home.

He and Mary had three children. Alonzo Church, Jr., was born in 1929, Mary Ann in 1933, and Mildred in 1938. (Mary Ann later married the logician John Addison.)

Alonzo Church had the polite manners of a gentleman who had grown up in Virginia. He was never known to be rude, even with people with whom he had strong disagreements. A deeply religious person, he was a lifelong member of the Presbyterian church.
Haskell did not show particular interest in mathematics when at high school and when he graduated in 1916 he fully intended to study medicine. He entered Harvard College. A major influence on the direction that his studies took was the entry of the United States into World War I in the spring of 1917. Curry wanted to serve his country, and decided that he would be more likely to see action if he had a mathematics training rather than if he continued the pre-medical course he was on. The war, however, ended shortly after this (in November) and on 9 December 1918 Curry left the army.
Haskell Curry...

Curry now decided that he would look for a career in electrical engineering and he took a job with the General Electric Company which allowed him to study electrical engineering part-time at the Massachusetts Institute of Technology.

However he soon discovered that he had a different attitude to the others taking the courses, for he wanted to know why a result was correct when for everyone else it only mattered that it was correct.

If one imagines that from 1924 when Curry embarked on his doctorate in mathematics at Harvard he at last found the topic for him, then one would be mistaken. He was given a topic in the theory of differential equations by George Birkhoff but he began reading books on logic which seemed to him far more interesting that his research topic.
Readings and References


- Read pp. 616–618, in Scott.
Acknowledgments