

# **The Lambda Calculus: Meta-Theory**

**Type Theory and Mechanized Reasoning**  
**Lecture 11**

# Introduction

# Administrivia

Homework 4 is due on *Thursday by 11:59PM*.

Homework 5 will be released on Friday (it will be short).

# Objectives

Finish our discussion on the `operational semantics` of the lambda calculus.

Introduce `semantic notions` of the lambda calculus.

Demonstrate how to `encode` data.

If we have time, use `De Bruijn indices` to avoid issues of  $\alpha$ -equivalence.

# Recap

# Recall: Lambda Terms

(Fix a set of variables.)

# Recall: Lambda Terms

(Fix a set of variables.)

**Definition.** The collection of **lambda terms** is defined inductively.

# Recall: Lambda Terms

(Fix a set of variables.)

**Definition.** The collection of **lambda terms** is defined inductively.

- Every variable  $x$  is a lambda term.

variables



# Recall: Lambda Terms

(Fix a set of variables.)

**Definition.** The collection of **lambda terms** is defined inductively.

- Every variable  $x$  is a lambda term.
- If  $M$  and  $N$  are lambda terms, then so is  $(MN)$

variables

application

# Recall: Lambda Terms

(Fix a set of variables.)

**Definition.** The collection of **lambda terms** is defined inductively.

- Every variable  $x$  is a lambda term.
- If  $M$  and  $N$  are lambda terms, then so is  $(MN)$
- If  $M$  is a lambda term, then so is  $(\lambda x.M)$  for any variable  $x$

variables

application

abstraction

# Examples (Again)

$x, y$

$$I \triangleq \lambda x . x$$

$$K \triangleq \lambda x . \lambda y . x$$

$$A \triangleq \lambda x . \lambda y . xy$$

$$\omega \triangleq \lambda x . xx$$

$$\Omega \triangleq \omega\omega = (\lambda x . xx)(\lambda x . xx)$$

# Evaluation (High Level)

$(\lambda x. \text{hat-on}(x))\text{cat}$

should evaluate to

$\text{hat-on}(\text{cat})$

# Evaluation (High Level)

$(\lambda x. \text{hat-on}(x))\text{cat}$

should evaluate to

$\text{hat-on}(\text{cat})$

We need to be able to **replace** the variable  $x$  in  $\text{hat-on}(x)$  with the argument to the function.

# Evaluation (High Level)

$(\lambda x. \text{hat-on}(x))\text{cat}$   
should evaluate to  
 $\text{hat-on}(\text{cat})$

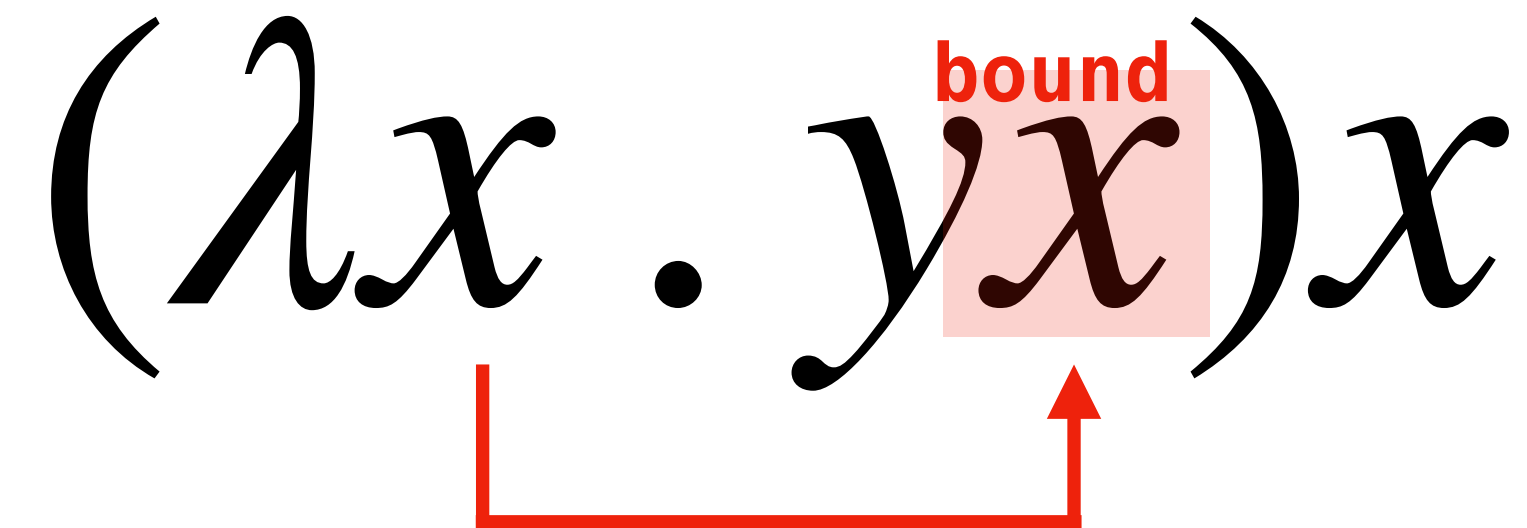
We need to be able to **replace** the variable  $x$  in  $\text{hat-on}(x)$  with the argument to the function.

The variable  $x$  is able to be replaced in  $\text{hat-on}(x)$  because it is not **bound** by anything.

# Recall: Free and Bound Variables

$$(\lambda x. yx)x$$

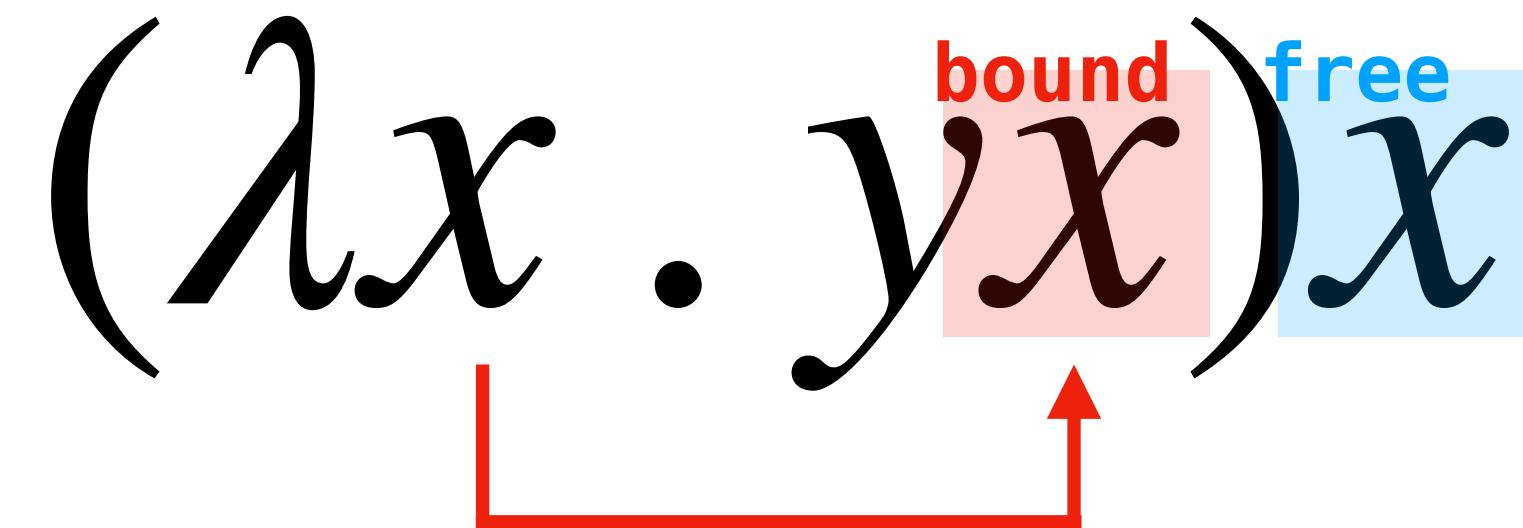
# Recall: Free and Bound Variables

$$(\lambda x. yx)x$$


A variable  $x$  is **bound** if it appears in the body of an abstraction over  $x$ .



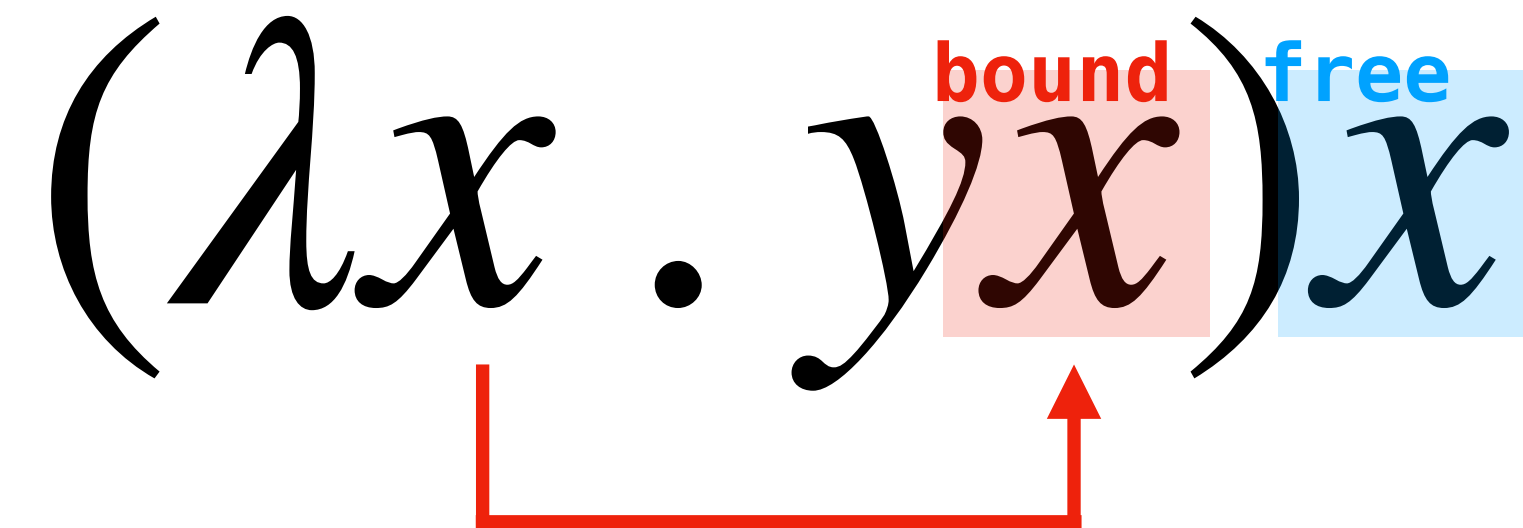
# Recall: Free and Bound Variables



A variable  $x$  is **bound** if it appears in the body of an abstraction over  $x$ .

Otherwise it is **free**.

# Recall: Free and Bound Variables



A variable  $x$  is **bound** if it appears in the body of an abstraction over  $x$ .

Otherwise it is **free**.

**Definition.** A term is **closed** if it has no free variables. Such a term is called a **combinator**.

# Recall: Substitution

# Recall: Substitution

**Definition.** Substitution of  $N$  for a free variable  $x$  in  $M$ , written  $M[N/x]$  is defined recursively on  $M$ .

# Recall: Substitution

**Definition.** Substitution of  $N$  for a free variable  $x$  in  $M$ , written  $M[N/x]$  is defined recursively on  $M$ .

- $y[N/x] = \begin{cases} N & y = x \\ y & \text{otherwise} \end{cases}$

# Recall: Substitution

**Definition.** Substitution of  $N$  for a free variable  $x$  in  $M$ , written  $M[N/x]$  is defined recursively on  $M$ .

- $y[N/x] = \begin{cases} N & y = x \\ y & \text{otherwise} \end{cases}$
- $(M_1M_2)[N/x] = (M_1[N/x])(M_2[N/x])$

# Recall: Substitution

**Definition.** Substitution of  $N$  for a free variable  $x$  in  $M$ , written  $M[N/x]$  is defined recursively on  $M$ .

- $y[N/x] = \begin{cases} N & y = x \\ y & \text{otherwise} \end{cases}$
- $(M_1 M_2)[N/x] = (M_1[N/x])(M_2[N/x])$
- $(\lambda y . M)[N/x] = \begin{cases} \lambda y . M & y = x \\ \lambda y . M[N/x] & \text{otherwise} \end{cases}$

# Recall: Substitution

**Definition.** Substitution of  $N$  for a free variable  $x$  in  $M$ , written  $M[N/x]$  is defined recursively on  $M$ .

- $y[N/x] = \begin{cases} N & y = x \\ y & \text{otherwise} \end{cases}$
- $(M_1 M_2)[N/x] = (M_1[N/x])(M_2[N/x])$
- $(\lambda y . M)[N/x] = \begin{cases} \lambda y . M & y = x \\ \lambda y . M[N/x] & \text{otherwise} \end{cases}$   
this is not quite right.



# Recall: Alpha-Equivalence

$$\lambda x . xz =_{\alpha} \lambda y . yz \neq_{\alpha} \lambda z . zz$$

# Recall: Alpha-Equivalence

$$\lambda x . xz =_{\alpha} \lambda y . yz \neq_{\alpha} \lambda z . zz$$

**Definition (Informal).** Lambda terms  $M$  and  $N$  are  $\alpha$ -equivalent if they are the same up to *valid* renaming bound variables.

# Recall: Alpha-Equivalence

$$\lambda x . xz =_{\alpha} \lambda y . yz \neq_{\alpha} \lambda z . zz$$

**Definition (Informal).** Lambda terms  $M$  and  $N$  are  $\alpha$ -equivalent if they are the same up to *valid* renaming bound variables.

*We cannot rename bound variables to names of existing free variables.*

# Recall: Alpha-Equivalence

$$\lambda x . xz =_{\alpha} \lambda y . yz \neq_{\alpha} \lambda z . zz$$

**Definition (Informal).** Lambda terms  $M$  and  $N$  are  $\alpha$ -equivalent if they are the same up to *valid* renaming bound variables.

*We cannot rename bound variables to names of existing free variables.*

**We will always consider terms up to  $\alpha$ -equivalence.**

moving on...

# Captured Variables

$$M =_{\alpha} M'$$

**should imply**

$$M[N/x] =_{\alpha} M'[N/x]$$

# Captured Variables

$$M =_{\alpha} M'$$

**should imply**

$$M[N/x] =_{\alpha} M'[N/x]$$

If we consider terms up to  $=_{\alpha}$ , then substitution should preserve  $=_{\alpha}$ .

# Captured Variables

$$M =_{\alpha} M'$$

**should imply**

$$M[N/x] =_{\alpha} M'[N/x]$$

If we consider terms up to  $=_{\alpha}$ , then substitution should preserve  $=_{\alpha}$ .

**Our current definition doesn't do this.**



# Captured Variables

$$\lambda x . y =_{\alpha} \lambda z . y$$

**but**

$$(\lambda x . y)[x/y] \neq_{\alpha} (\lambda z . y)[x/y]$$

# Captured Variables

$$\lambda x . y =_{\alpha} \lambda z . y$$

**but**

$$(\lambda x . y)[x/y] \neq_{\alpha} (\lambda z . y)[x/y]$$

Since  $x$  appears free in the value being substituted in, it will be **captured**.

# Substitution (Again)

**Definition.** Substitution of  $N$  for  $x$  in  $M$ , written  $M[N/x]$  is defined recursively on  $M$ .

- $y[N/x] = \begin{cases} N & y = x \\ y & \text{otherwise} \end{cases}$
- $(M_1 M_2)[N/x] = (M_1[N/x])(M_2[N/x])$
- $(\lambda y . M)[N/x] = \begin{cases} \lambda y . M & y = x \\ (\lambda z . M[z/y])[N/x] & \text{otherwise} \end{cases}$   
where  $z$  does not appear free in  $M$  or  $N$

# Fresh Variables

# Fresh Variables

**Definition.** A variable  $x$  is **fresh** with respect to a term  $M$  if  $x$  does not appear free in  $M$ .

# Fresh Variables

**Definition.** A variable  $x$  is **fresh** with respect to a term  $M$  if  $x$  does not appear free in  $M$ .

A couple ways to deal with it:

# Fresh Variables

**Definition.** A variable  $x$  is **fresh** with respect to a term  $M$  if  $x$  does not appear free in  $M$ .

A couple ways to deal with it:

- » Keep track of a counter (this requires state)
- » If variables are indexed, take the max of all indices + 1
- » Use De Bruijn indices to avoid this all together

# Fresh Variables

**Definition.** A variable  $x$  is **fresh** with respect to a term  $M$  if  $x$  does not appear free in  $M$ .

A couple ways to deal with it:

- » Keep track of a counter (this requires state)
- » If variables are indexed, take the max of all indices + 1
- » Use De Bruijn indices to avoid this all together

(Finding fresh variables is more difficult in the functional setting.)



# Reduction

# Beta Reduction

# Beta Reduction

**Definition.** We define the relation  $M \rightarrow_{\beta} N$  as follows.

# Beta Reduction

**Definition.** We define the relation  $M \rightarrow_{\beta} N$  as follows.

- $(\lambda x . M)N \rightarrow_{\beta} M[N/x]$

# Beta Reduction

**Definition.** We define the relation  $M \rightarrow_{\beta} N$  as follows.

- $(\lambda x . M)N \rightarrow_{\beta} M[N/x]$
- $M \rightarrow_{\beta} M'$  implies  $MN \rightarrow_{\beta} M'N$  and  $NM \rightarrow_{\beta} NM'$  and  $\lambda x . M \rightarrow_{\beta} \lambda x . M'$

# Beta Reduction

**Definition.** We define the relation  $M \rightarrow_{\beta} N$  as follows.

- $(\lambda x . M)N \rightarrow_{\beta} M[N/x]$
- $M \rightarrow_{\beta} M'$  implies  $MN \rightarrow_{\beta} M'N$  and  $NM \rightarrow_{\beta} NM'$  and  $\lambda x . M \rightarrow_{\beta} \lambda x . M'$

This is a **relation** not a function.

# Reduction

# Reduction

**Definition.**  $\rightarrow_{\beta}$  for the reflexive transitive closure of  $\rightarrow_{\beta}$ .



# Reduction

**Definition.**  $\rightarrow_{\beta}$  for the reflexive transitive closure of  $\rightarrow_{\beta}$ .

That is,  $M \rightarrow_{\beta} N$  if there is a (possibly empty) sequence of reductions

# Reduction

**Definition.**  $\rightarrow_{\beta}$  for the reflexive transitive closure of  $\rightarrow_{\beta}$ .

That is,  $M \rightarrow_{\beta} N$  if there is a (possibly empty) sequence of reductions

$$M = P_1 \rightarrow_{\beta} P_2 \dots \rightarrow_{\beta} P_k = N$$

# Reduction

**Definition.**  $\rightarrow_{\beta}$  for the reflexive transitive closure of  $\rightarrow_{\beta}$ .

That is,  $M \rightarrow_{\beta} N$  if there is a (possibly empty) sequence of reductions

$$M = P_1 \rightarrow_{\beta} P_2 \dots \rightarrow_{\beta} P_k = N$$

This captures what happens when we "compute" a lambda term.

# Redex

$$\dots((\lambda x . M)N)\dots \rightarrow_{\beta} \dots(M[N/x])\dots$$

# Redex

$$\dots((\lambda x . M)N)\dots \rightarrow_{\beta} \dots(M[N/x])\dots$$

**Definition.** A subterm is a **redex** if it is of the form.

# Redex

$$\dots((\lambda x . M)N)\dots \rightarrow_{\beta} \dots(M[N/x])\dots$$

**Definition.** A subterm is a **redex** if it is of the form.

A term may have many redexes, which means there may be **multiple ways to  $\beta$ -reduce a term.**

# Normal Forms

# Normal Forms

**Definition.** A  $\beta$ -normal form is a term  $M$  such that there is no  $N$  where  $M \rightarrow_{\beta} N$ .



# Normal Forms

**Definition.** A  $\beta$ -normal form is a term  $M$  such that there is no  $N$  where  $M \rightarrow_{\beta} N$ .

Normal forms cannot be further reduced, they are like the **values** of a computation.

# Normal Forms

**Definition.** A  $\beta$ -normal form is a term  $M$  such that there is no  $N$  where  $M \rightarrow_{\beta} N$ .

Normal forms cannot be further reduced, they are like the **values** of a computation.

**Examples.**  $\lambda x.x$ ,  $\lambda x.\lambda y.x$ ,  $\lambda x.xx$ , are normal forms whereas  $(\lambda x.x)(\lambda x.x)$  is not.

# Meta-Theory

# Meta-Theoretic Questions

- Do all terms have normal forms?
- If a term has a normal form, is it unique?
- If a term has a normal form, is there always a way to find it?

# Meta-Theoretic Questions

- Do all terms have normal forms?
- If a term has a normal form, is it unique?
- If a term has a normal form, is there always a way to find it?

# Meta-Theoretic Questions

- Do all terms have normal forms?
- If a term has a normal form, is it unique?
- If a term has a normal form, is there always a way to find it?

# Omega Combinator

$$\Omega = \omega\omega = (\lambda x . xx)(\lambda x . xx)$$

# Omega Combinator

$$\Omega = \omega\omega = (\lambda x . xx)(\lambda x . xx)$$

This is the `infinite-loop` combinator:



# Omega Combinator

$$\Omega = \omega\omega = (\lambda x . xx)(\lambda x . xx)$$

This is the `infinite-loop` combinator:

$$(\lambda x . xx)(\lambda x . xx) \rightarrow_{\beta} (xx)[\lambda x . xx/x] = (\lambda x . xx)(\lambda x . xx)$$

# Omega Combinator

$$\Omega = \omega\omega = (\lambda x . xx)(\lambda x . xx)$$

This is the `infinite-loop` combinator:

$$(\lambda x . xx)(\lambda x . xx) \rightarrow_{\beta} (xx)[\lambda x . xx/x] = (\lambda x . xx)(\lambda x . xx)$$

So this term does not have a normal form.

# Meta-Theoretic Questions

- Do all terms have normal forms?
- If a term has a normal form, is it unique?
- If a term has a normal form, is there always a way to find it?

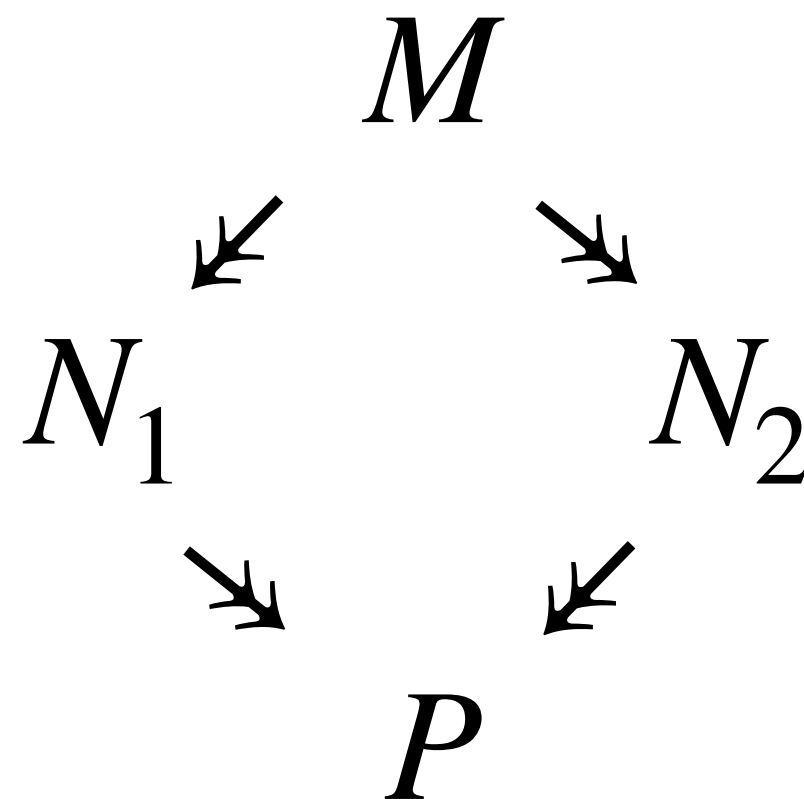
# Meta-Theoretic Questions

- Do all terms have normal forms?
- If a term has a normal form, is it unique?
- If a term has a normal form, is there always a way to find it?

# Confluence (Church-Rosser)

**Theorem.** If  $M \twoheadrightarrow_{\beta} N_1$  and  $M \twoheadrightarrow_{\beta} N_2$  then there is a term  $P$  such that  $N_1 \twoheadrightarrow_{\beta} P$  and  $N_2 \twoheadrightarrow_{\beta} P$ .

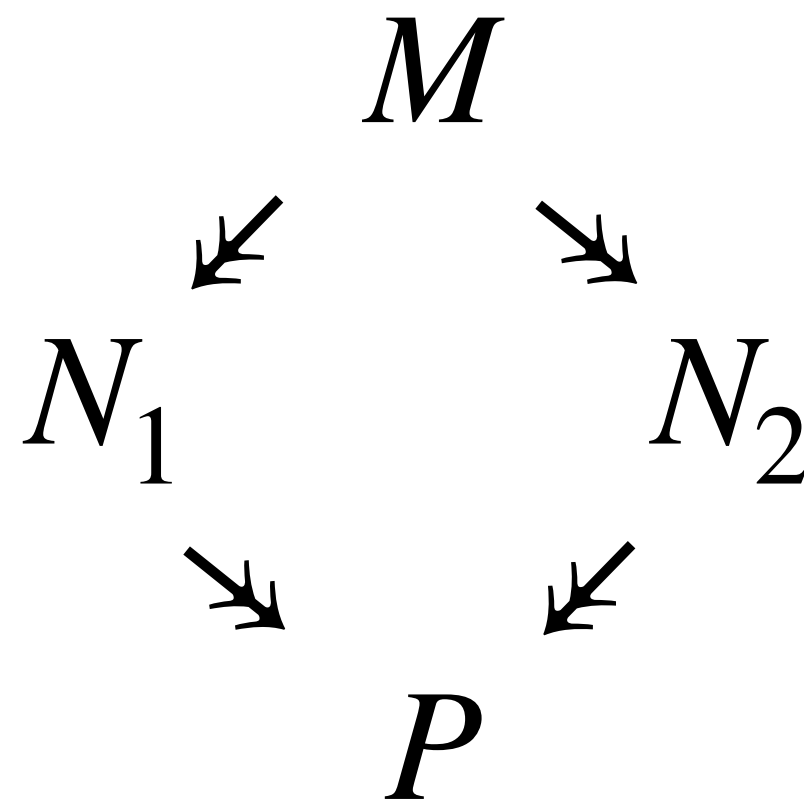
We can't "completely diverge" after reducing.



# Confluence (Church-Rosser)

**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  $N_2 \rightarrow_{\beta} P$ .

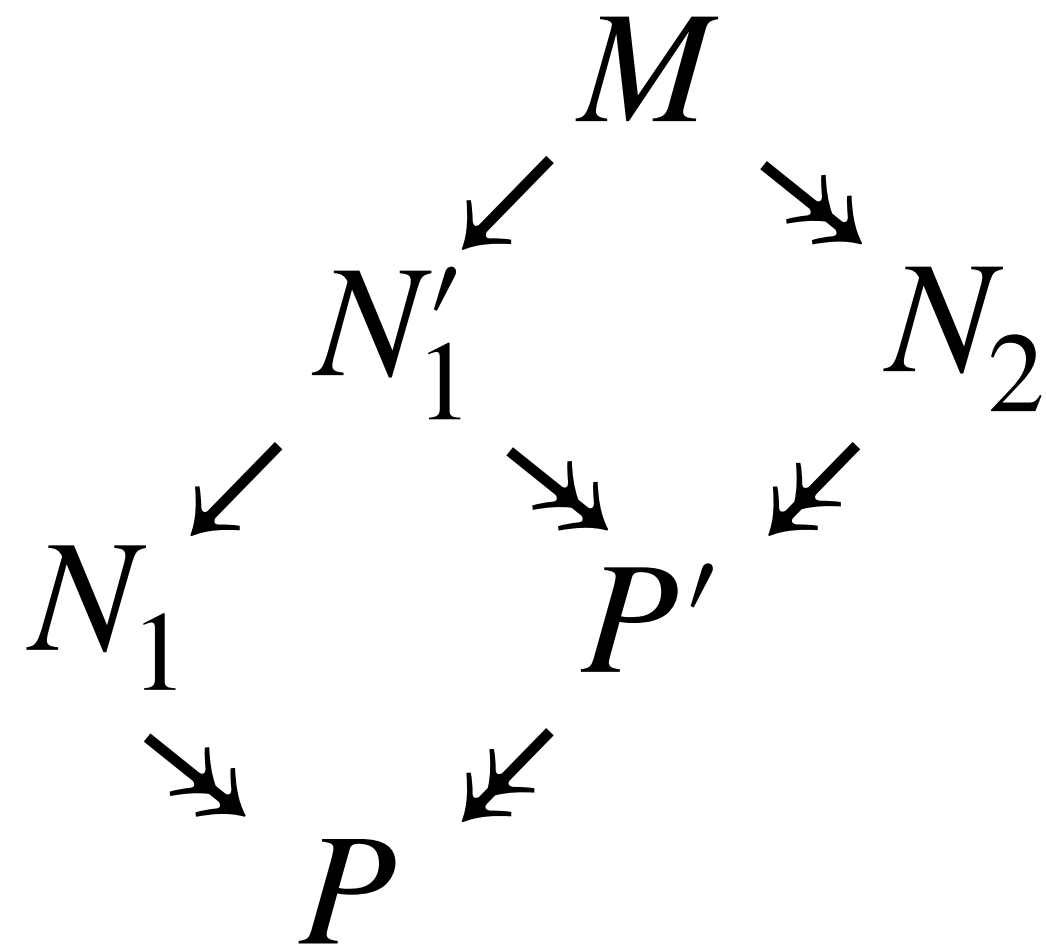
We can't "completely diverge" after reducing.



# Confluence: Very Rough Proof Sketch

**Theorem.** If  $M \twoheadrightarrow_{\beta} N_1$  and  $M \twoheadrightarrow_{\beta} N_2$  then there is a term  $P$  such that  $N_1 \twoheadrightarrow_{\beta} P$  and  $N_2 \twoheadrightarrow P$ .

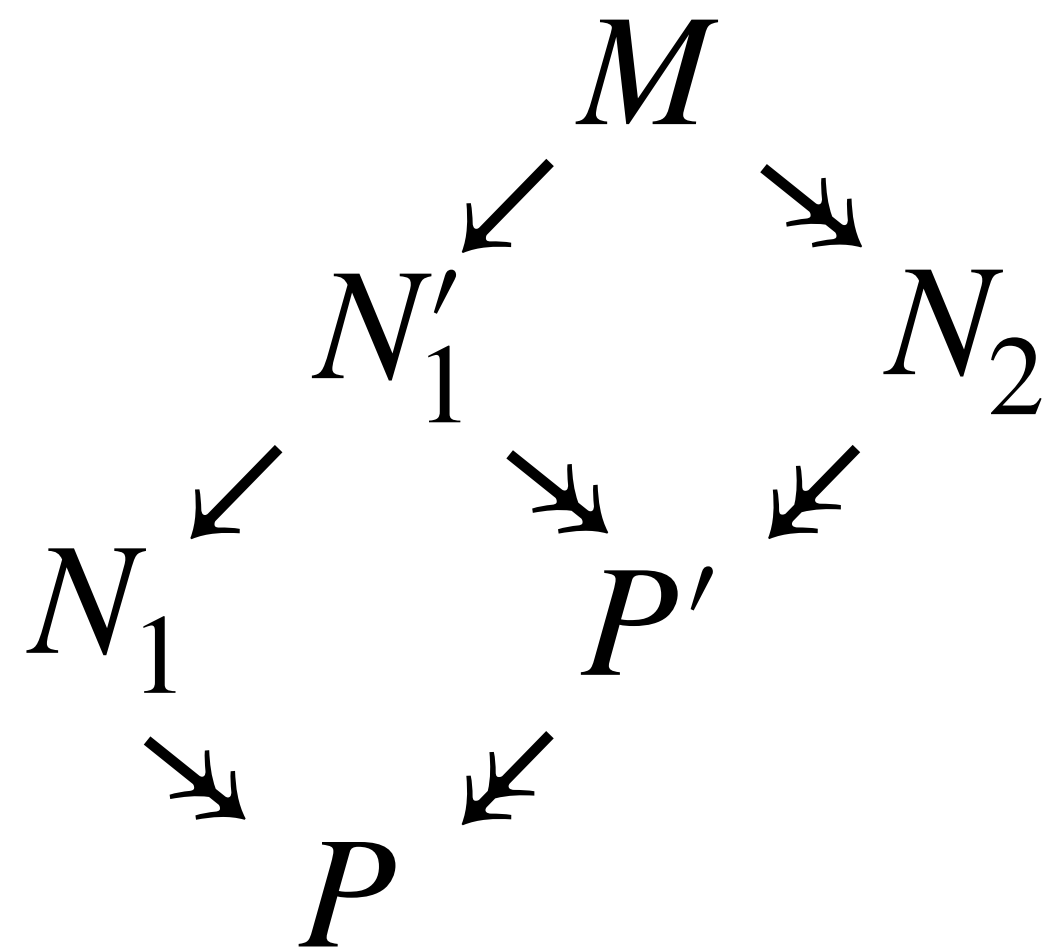
We "unravel" each step on the paths and fill in the parallelogram.



# Confluence: Very Rough Proof Sketch

**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  $N_2 \rightarrow_{\beta} P$ .

We "unravel" each step on the paths and fill in the parallelogram.

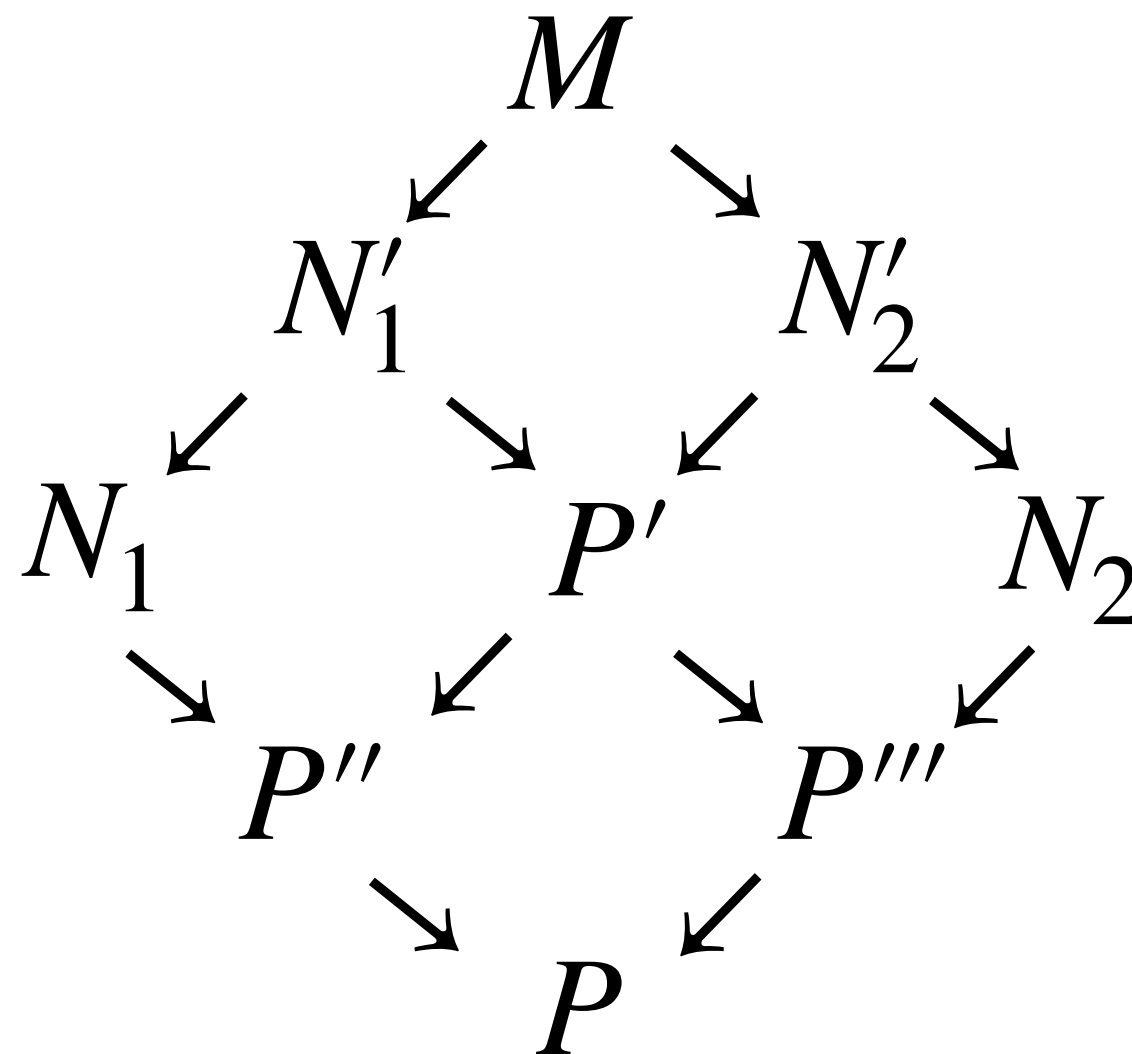




# Confluence: Very Rough Proof Sketch

**Theorem.** If  $M \twoheadrightarrow_{\beta} N_1$  and  $M \twoheadrightarrow_{\beta} N_2$  then there is a term  $P$  such that  $N_1 \twoheadrightarrow_{\beta} P$  and  $N_2 \twoheadrightarrow_{\beta} P$ .

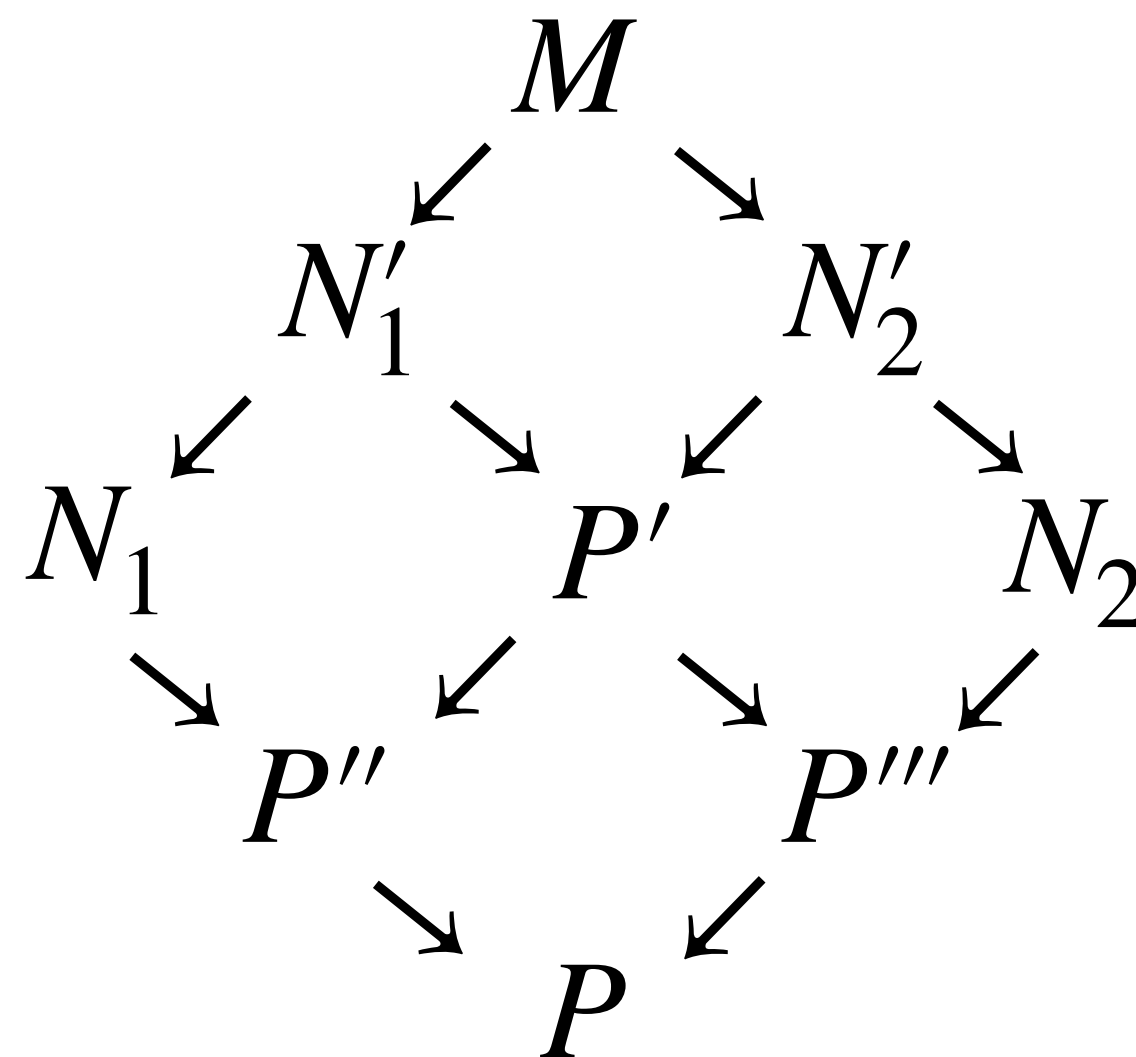
We "unravel" each step on the paths and fill in the parallelogram.



# Confluence: Very Rough Proof Sketch

**Theorem.** If  $M \twoheadrightarrow_{\beta} N_1$  and  $M \twoheadrightarrow_{\beta} N_2$  then there is a term  $P$  such that  $N_1 \twoheadrightarrow_{\beta} P$  and  $N_2 \twoheadrightarrow_{\beta} P$ .

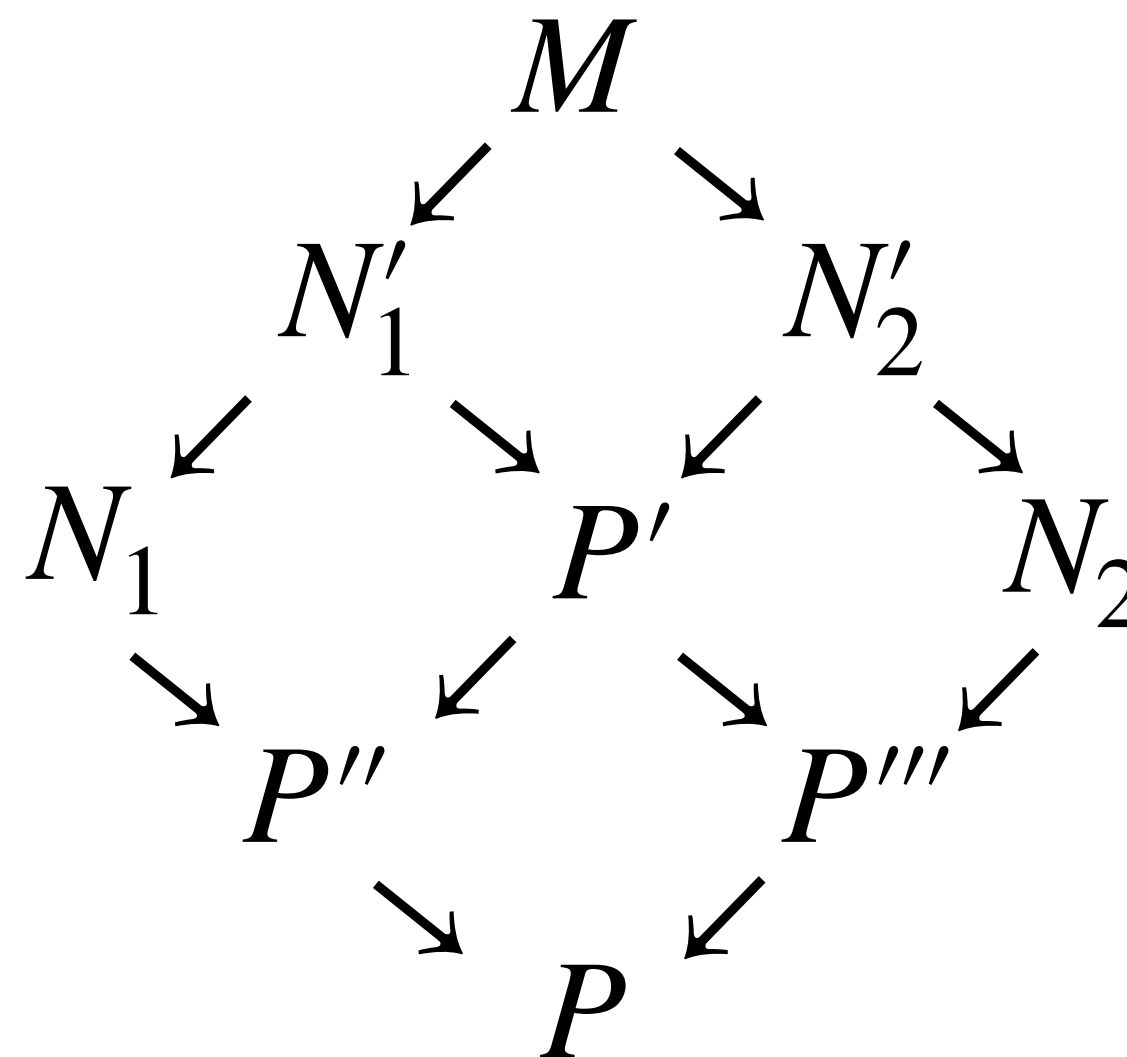
We "unravel" each step on the paths and fill in the parallelogram.



# Confluence: Very Rough Proof Sketch

**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  $N_2 \rightarrow_{\beta} P$ .

We "unravel" each step on the paths and fill in the parallelogram.



**It's a bit more complicated than this in reality.**

# Unique Normal Forms

**Theorem.** If  $M \rightarrow_{\beta} N$  and  $M \rightarrow_{\beta} P$  and  $N$  and  $P$  are normal forms, then  $N = P$ .

**Proof.** There is a term  $Z$  such that  $N \rightarrow_{\beta} Z$  and  $P \rightarrow_{\beta} Z$ . Since  $N$  and  $P$  are normal forms, it must be that

$$N = Z = P$$

# Unique Normal Forms

**Theorem.** If  $M \rightarrow_{\beta} N$  and  $M \rightarrow_{\beta} P$  and  $N$  and  $P$  are normal forms, then  $N = P$ .

**Proof.** There is a term  $Z$  such that  $N \rightarrow_{\beta} Z$  and  $P \rightarrow_{\beta} Z$ . Since  $N$  and  $P$  are normal forms, it must be that

$$N = Z = P$$

# Meta-Theoretic Questions

- Do all terms have normal forms?
- If a term has a normal form, is it unique?
- If a term has a normal form, is there always a way to find it?

# Meta-Theoretic Questions

- Do all terms have normal forms?
- If a term has a normal form, is it unique?
- If a term has a normal form, is there always a way to find it?

# Meta-Theoretic Questions

- Do all terms have normal forms?
- If a term has a normal form, is it unique?
- If a term has a normal form, is there always a way to find it?

**This is a subtle question...**



# Evaluation

# Evaluation

Evaluation is the process of **reducing** a term to a **normal form**, if possible. But we have to worry about the order in which we choose redexes.

# Evaluation

Evaluation is the process of **reducing** a term to a **normal form**, if possible. But we have to worry about the order in which we choose redexes.

**Definition (Informal).** A (one-step) **evaluation strategy** is a way of determining which redexes to reduce.

# Evaluation

Evaluation is the process of **reducing** a term to a **normal form**, if possible. But we have to worry about the order in which we choose redexes.

**Definition (Informal).** A (one-step) **evaluation strategy** is a way of determining which redexes to reduce.

We apply our strategy over and over until we reach a normal form (or run forever).

# Two Common Evaluation Strategies

# Two Common Evaluation Strategies

**Leftmost outermost (Normal).** Reduce the leftmost redex that does not appear in any other redexes.

# Two Common Evaluation Strategies

**Leftmost outermost (Normal).** Reduce the leftmost redex that does not appear in any other redexes.

$$(\lambda x . xx)((\lambda x . x)y) \rightarrow_{\beta} ((\lambda x . x)y)((\lambda x . x)y)$$

# Two Common Evaluation Strategies

**Leftmost outermost (Normal).** Reduce the leftmost redex that does not appear in any other redexes.

$$(\lambda x . xx)((\lambda x . x)y) \rightarrow_{\beta} ((\lambda x . x)y)((\lambda x . x)y)$$

**Leftmost innermost (Applicative).** Reduce the leftmost redex which does not contain any other redexes.



# Two Common Evaluation Strategies

**Leftmost outermost (Normal).** Reduce the leftmost redex that does not appear in any other redexes.

$$(\lambda x . xx)((\lambda x . x)y) \rightarrow_{\beta} ((\lambda x . x)y)((\lambda x . x)y)$$

**Leftmost innermost (Applicative).** Reduce the leftmost redex which does not contain any other redexes.

$$(\lambda x . xx)((\lambda x . x)y) \rightarrow_{\beta} (\lambda x . xx)y$$

# Normal Order

# Normal Order

A function is *immediately applied* to its argument.

# Normal Order

A function is **immediately applied** to its argument.

This roughly corresponds to "Call-by-Name" evaluation (though not exactly).

# Normal Order

A function is **immediately applied** to its argument.

This roughly corresponds to "Call-by-Name" evaluation (though not exactly).

**Theorem.** If  $M \rightarrow_{\beta}^* N$  and  $N$  is a normal form, then the normal order reduction strategy reduces  $M$  to  $N$ .

# Normal Order

A function is **immediately applied** to its argument.

This roughly corresponds to "Call-by-Name" evaluation (though not exactly).

**Theorem.** If  $M \rightarrow_{\beta} N$  and  $N$  is a normal form, then the normal order reduction strategy reduces  $M$  to  $N$ .

**Example.**  $(\lambda f. \lambda x. f(fx))((\lambda x. \lambda y. x)z)$  (on the board)

# Applicative Order

# Applicative Order

The argument is fully **evaluated before** the function is called.



# Applicative Order

The argument is fully `evaluated before` the function is called.

This roughly corresponds to "Call-by-value".

# Applicative Order

The argument is fully **evaluated before** the function is called.

This roughly corresponds to "Call-by-value".

**This strategy may not terminate, even if there is a normal form:**

# Applicative Order

The argument is fully *evaluated before* the function is called.

This roughly corresponds to "Call-by-value".

**This strategy may not terminate, even if there is a normal form:**

$$K\Omega I = (\lambda x . \lambda y . x)\Omega(\lambda x . x)$$

# **Weak and Strong Normalization**

# Weak and Strong Normalization

**Definition.** A term is **weakly normalizing** if it has a normal form.

# Weak and Strong Normalization

**Definition.** A term is **weakly normalizing** if it has a normal form.

**Definition.** A term is **strongly normalizing** if it has no infinite reduction sequences.

# Weak and Strong Normalization

**Definition.** A term is **weakly normalizing** if it has a normal form.

**Definition.** A term is **strongly normalizing** if it has no infinite reduction sequences.

Strong normalization means we (barring complexity concerns) we **don't need to think about** the reduction strategy.

# Weak and Strong Normalization

**Definition.** A term is **weakly normalizing** if it has a normal form.

**Definition.** A term is **strongly normalizing** if it has no infinite reduction sequences.

Strong normalization means we (barring complexity concerns) we **don't need to think about** the reduction strategy.

**Examples.**  $(\lambda x.x)(\lambda x.x)$  is SN.  $K\Omega I$  is WN but not SN.  $\Omega$  is neither.



# Encoding

# Church Booleans

$$\text{tru} = \lambda x . \lambda y . x$$

$$\text{fls} = \lambda x . \lambda y . y$$

Booleans are represented as *computations* which, given two values, chooses one based on the Boolean value we're representing.

***Question.*** Can we implement ***if-then-else***?

# Church Numerals

$$\text{zero} = \lambda f . \lambda x . x$$

$$\text{one} = \lambda f . \lambda x . fx$$

$$\text{two} = \lambda f . \lambda x . f(fx)$$

$$\text{suc} = \lambda n . \lambda f . \lambda x . nf(fx)$$

Numbers can be represented as "folds" or "recursors". Given a function  $f$  and a base value  $k$ ,  $n$  is represented by the computation that applies  $f$  to  $k$  a total of  $n$  times.

***Question.*** Can we implement ***add***?

# Computability and the Lambda Calculus

**Theorem (Informal).** The lambda calculus is Turing-complete.

Any partial function on numbers which can be written as a Turing Machine (or a Python program) can be written as a lambda term on Church numerals.

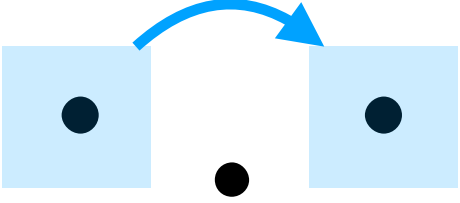
# Computability and the Lambda Calculus

**Theorem (Informal).** The lambda calculus is Turing-complete.

Any partial function on numbers which can be written as a Turing Machine (or a Python program) can be written as a lambda term on Church numerals.

# De Bruijn Indices

# Motivation

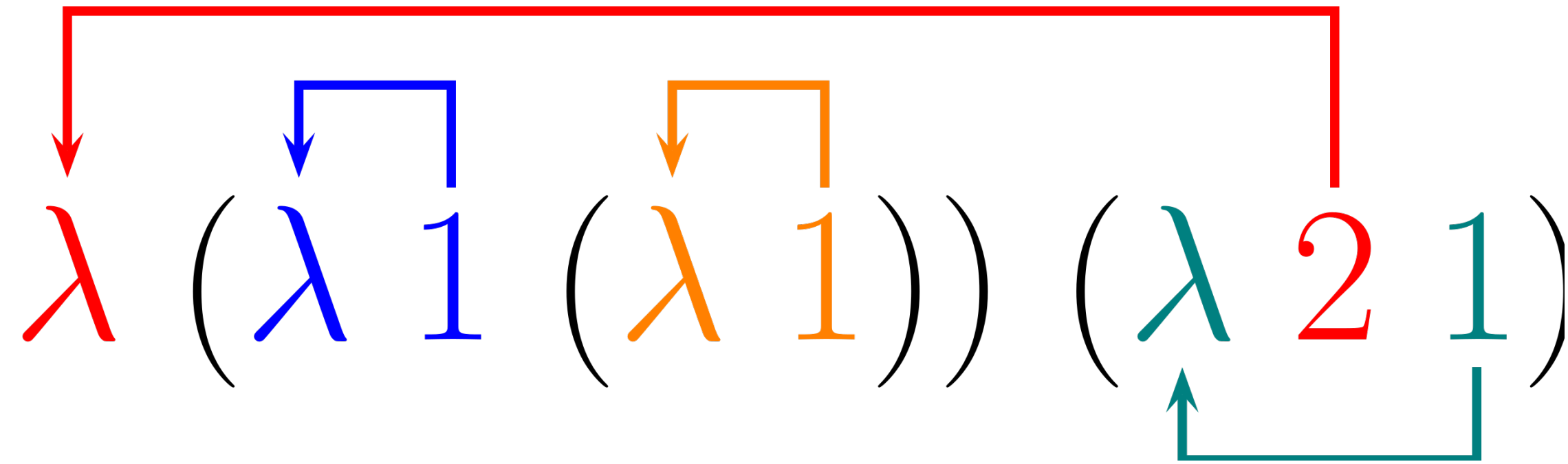
$$\lambda x . xz =_{\alpha} \lambda y . yz \neq_{\alpha} \lambda \boxed{\bullet} . \boxed{\bullet} z$$


We always consider terms up to  $=_{\alpha}$ .

What we *really* want is to be able to replace the binding variable with a **pointer**.

In math speak, we want to give a "canonical element" for the  $\alpha$ -equivalence class.

# De Bruijn Indices



*The idea.* Bound variables are represented as numbers, the depth away from the *binding site*.

$$M ::= \mathbb{N} \mid \lambda M \mid MM$$

This gives an incredibly simple grammar.



# What about free variables?

We can use numbers larger than the depth of the formula.

$$\lambda x . xz \implies \lambda(1\ 2)$$

Or we can use the "locally nameless representation":

$$\lambda x . xz \implies \lambda(1\ z)$$

We keep free variables as they are, and use De Bruijn indices for bound variables.

# Pros and Cons

- We no longer need to consider  $=_{\alpha}$ , equality is structural equality
- $\beta$ -reduction is a bit harder, now we need to update De-Bruijn indices at each step.
- They make terms harder to read.