

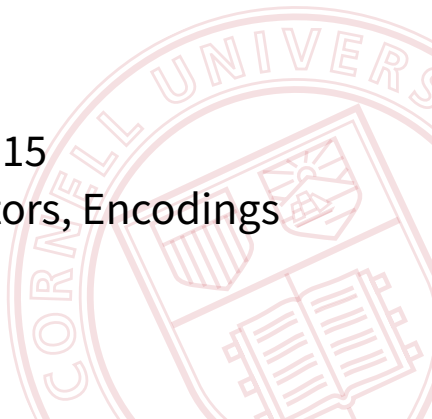
CS 4110

# Programming Languages & Logics

---

Lecture 15

De Bruijn, Combinators, Encodings



# Review: $\lambda$ -calculus

## Syntax

$$\begin{aligned} e &::= x \mid e_1 e_2 \mid \lambda x. e \\ v &::= \lambda x. e \end{aligned}$$

## Semantics

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \qquad \frac{e \rightarrow e'}{v e \rightarrow v e'}$$

$$\frac{}{(\lambda x. e) v \rightarrow e\{v/x\}} \beta$$

# Rewind: Currying

This is just a function that returns a function:

$$\text{ADD} \triangleq \lambda x. \lambda y. x + y$$

$$\text{ADD } 38 \rightarrow \lambda y. 38 + y$$

$$\text{ADD } 38 \ 4 = (\text{ADD } 38) \ 4 \rightarrow 42$$

**Informally**, you can think of it as a *curried* function that takes two arguments, one after the other.

But that's just a way to get intuition. The  $\lambda$ -calculus only has one-argument functions.

# de Bruijn Notation

Another way to avoid the tricky issues with substitution is to use a *nameless* representation of terms.

$$e ::= n \mid \lambda.e \mid e e$$

# de Bruijn Notation

Another way to avoid the tricky issues with substitution is to use a *nameless* representation of terms.

$$e ::= n \mid \lambda.e \mid e e$$

Abstractions have lost their variables!

Variables are replaced with numerical indices!

# Examples

---

Here are some terms written in standard and de Bruijn notation:

**Standard**

$\lambda x. x$

$\lambda z. z$

**de Bruijn**

$\lambda. 0$

# Examples

Here are some terms written in standard and de Bruijn notation:

**Standard**

$\lambda x. x$

$\lambda z. z$

$\lambda x. \lambda y. x$

**de Bruijn**

$\lambda. 0$

$\lambda. 0$

# Examples

Here are some terms written in standard and de Bruijn notation:

## Standard

$\lambda x. x$

$\lambda z. z$

$\lambda x. \lambda y. x$

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

## de Bruijn

$\lambda. 0$

$\lambda. 0$

$\lambda. \lambda. 1$



# Examples

Here are some terms written in standard and de Bruijn notation:

## Standard

$\lambda x. x$

$\lambda z. z$

$\lambda x. \lambda y. x$

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

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

## de Bruijn

$\lambda. 0$

$\lambda. 0$

$\lambda. \lambda. 1$

$\lambda. \lambda. \lambda. \lambda. 3\ 1\ (2\ 1\ 0)$

# Examples

Here are some terms written in standard and de Bruijn notation:

## Standard

$\lambda x. x$

$\lambda z. z$

$\lambda x. \lambda y. x$

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

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

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

## de Bruijn

$\lambda. 0$

$\lambda. 0$

$\lambda. \lambda. 1$

$\lambda. \lambda. \lambda. \lambda. 3\ 1\ (2\ 1\ 0)$

$(\lambda. 0\ 0) (\lambda. 0\ 0)$

# Examples

Here are some terms written in standard and de Bruijn notation:

## Standard

$\lambda x. x$

$\lambda z. z$

$\lambda x. \lambda y. x$

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

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

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

## de Bruijn

$\lambda. 0$

$\lambda. 0$

$\lambda. \lambda. 1$

$\lambda. \lambda. \lambda. \lambda. 3\ 1\ (2\ 1\ 0)$

$(\lambda. 0\ 0) (\lambda. 0\ 0)$

$(\lambda. \lambda. 0) (\lambda. 0)$

# Free variables

To represent a  $\lambda$ -expression that contains free variables in de Bruijn notation, we need a way to map the free variables to integers.

We will work with respect to a map  $\Gamma$  from variables to integers called a *context*.

## Examples:

Suppose that  $\Gamma$  maps  $x$  to 0 and  $y$  to 1.

- Representation of  $xy$  is 0 1
- Representation of  $\lambda z. xyz \lambda. 1 2 0$

# Shifting

To define substitution, we will need an operation that shifts by  $i$  the variables above a cutoff  $c$ :

$$\begin{aligned}\uparrow_c^i(n) &= \begin{cases} n & \text{if } n < c \\ n + i & \text{otherwise} \end{cases} \\ \uparrow_c^i(\lambda.e) &= \lambda.(\uparrow_{c+1}^i e) \\ \uparrow_c^i(e_1 e_2) &= (\uparrow_c^i e_1) (\uparrow_c^i e_2)\end{aligned}$$

The cutoff  $c$  keeps track of the variables that were bound in the original expression and so should not be shifted.

The cutoff is 0 initially.

# Substitution

Now we can define substitution:

$$\begin{aligned}n\{e/m\} &= \begin{cases} e & \text{if } n = m \\ n & \text{otherwise} \end{cases} \\(\lambda.e_1)\{e/m\} &= \lambda.e_1\{(\uparrow_0^1 e)/m + 1\} \\(e_1 e_2)\{e/m\} &= (e_1\{e/m\}) (e_2\{e/m\})\end{aligned}$$

# Substitution

Now we can define substitution:

$$\begin{aligned}n\{e/m\} &= \begin{cases} e & \text{if } n = m \\ n & \text{otherwise} \end{cases} \\(\lambda.e_1)\{e/m\} &= \lambda.e_1\{(\uparrow_0^1 e)/m + 1\} \\(e_1 e_2)\{e/m\} &= (e_1\{e/m\}) (e_2\{e/m\})\end{aligned}$$

The  $\beta$  rule for terms in de Bruijn notation is just:

$$\overline{(\lambda.e_1) e_2 \rightarrow \uparrow_0^{-1} (e_1\{\uparrow_0^1 e_2/0\})} \beta$$

# Example

---

Consider the term  $(\lambda u. \lambda v. u \ x) \ y$  with respect to a context where  $\Gamma(x) = 0$  and  $\Gamma(y) = 1$ .



# Example

---

Consider the term  $(\lambda u. \lambda v. u \ x) \ y$  with respect to a context where  $\Gamma(x) = 0$  and  $\Gamma(y) = 1$ .

$$(\lambda. \lambda. 1 \ 2) \ 1$$

# Example

Consider the term  $(\lambda u. \lambda v. u x) y$  with respect to a context where  $\Gamma(x) = 0$  and  $\Gamma(y) = 1$ .

$$\begin{aligned} & (\lambda. \lambda. 1 \ 2) \ 1 \\ \rightarrow & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{(\uparrow_0^1 1)/0\}) \end{aligned}$$

# Example

Consider the term  $(\lambda u. \lambda v. u x) y$  with respect to a context where  $\Gamma(x) = 0$  and  $\Gamma(y) = 1$ .

$$\begin{aligned} & (\lambda. \lambda. 1 \ 2) \ 1 \\ \rightarrow & \ \uparrow_0^{-1} ((\lambda. 1 \ 2) \{(\uparrow_0^1 1)/0\}) \\ = & \ \uparrow_0^{-1} ((\lambda. 1 \ 2) \{2/0\}) \end{aligned}$$

# Example

Consider the term  $(\lambda u. \lambda v. u \ x) \ y$  with respect to a context where  $\Gamma(x) = 0$  and  $\Gamma(y) = 1$ .

$$\begin{aligned} & (\lambda. \lambda. 1 \ 2) \ 1 \\ \rightarrow & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{2/0\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{(\uparrow_0^1 2)/(0 + 1)\}) \end{aligned}$$

# Example

Consider the term  $(\lambda u. \lambda v. u x) y$  with respect to a context where  $\Gamma(x) = 0$  and  $\Gamma(y) = 1$ .

$$\begin{aligned} & (\lambda. \lambda. 1 \ 2) \ 1 \\ \rightarrow & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{2/0\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{(\uparrow_0^1 2)/(0 + 1)\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{3/1\}) \end{aligned}$$

# Example

Consider the term  $(\lambda u. \lambda v. u x) y$  with respect to a context where  $\Gamma(x) = 0$  and  $\Gamma(y) = 1$ .

$$\begin{aligned} & (\lambda. \lambda. 1 \ 2) \ 1 \\ \rightarrow & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{2/0\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{(\uparrow_0^1 2)/(0 + 1)\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{3/1\}) \\ = & \uparrow_0^{-1} \lambda. (1 \{3/1\}) (2 \{3/1\}) \end{aligned}$$

# Example

Consider the term  $(\lambda u. \lambda v. u x) y$  with respect to a context where  $\Gamma(x) = 0$  and  $\Gamma(y) = 1$ .

$$\begin{aligned} & (\lambda. \lambda. 1 \ 2) \ 1 \\ \rightarrow & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{2/0\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{(\uparrow_0^1 2)/(0 + 1)\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{3/1\}) \\ = & \uparrow_0^{-1} \lambda. (1 \{3/1\}) (2 \{3/1\}) \\ = & \uparrow_0^{-1} \lambda. 3 \ 2 \end{aligned}$$

# Example

Consider the term  $(\lambda u. \lambda v. u x) y$  with respect to a context where  $\Gamma(x) = 0$  and  $\Gamma(y) = 1$ .

$$\begin{aligned} & (\lambda. \lambda. 1 \ 2) \ 1 \\ \rightarrow & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{2/0\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{(\uparrow_0^1 2)/(0 + 1)\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{3/1\}) \\ = & \uparrow_0^{-1} \lambda. (1 \{3/1\}) (2 \{3/1\}) \\ = & \uparrow_0^{-1} \lambda. 3 \ 2 \\ = & \lambda. 2 \ 1 \end{aligned}$$



# Example

Consider the term  $(\lambda u. \lambda v. u x) y$  with respect to a context where  $\Gamma(x) = 0$  and  $\Gamma(y) = 1$ .

$$\begin{aligned} & (\lambda. \lambda. 1 \ 2) \ 1 \\ \rightarrow & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{(\uparrow_0^1 1)/0\}) \\ = & \uparrow_0^{-1} ((\lambda. 1 \ 2) \{2/0\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{(\uparrow_0^1 2)/(0 + 1)\}) \\ = & \uparrow_0^{-1} \lambda. ((1 \ 2) \{3/1\}) \\ = & \uparrow_0^{-1} \lambda. (1 \{3/1\}) (2 \{3/1\}) \\ = & \uparrow_0^{-1} \lambda. 3 \ 2 \\ = & \lambda. 2 \ 1 \end{aligned}$$

which, in standard notation (with respect to  $\Gamma$ ), is the same as  $\lambda v. y x$ .

# Combinators

---

Another way to avoid the issues having to do with free and bound variable names in the  $\lambda$ -calculus is to work with closed expressions or *combinators*.

With just three combinators, we can encode the entire  $\lambda$ -calculus.

# Combinators

Another way to avoid the issues having to do with free and bound variable names in the  $\lambda$ -calculus is to work with closed expressions or *combinators*.

With just three combinators, we can encode the entire  $\lambda$ -calculus.

$$K = \lambda x. \lambda y. x$$

$$S = \lambda x. \lambda y. \lambda z. x z (y z)$$

$$I = \lambda x. x$$

# Combinators

We can even define independent evaluation rules that don't depend on the  $\lambda$ -calculus at all.

Behold the “SKI-calculus”:

$$K\ e_1\ e_2 \rightarrow e_1$$

$$S\ e_1\ e_2\ e_3 \rightarrow e_1\ e_3\ (e_2\ e_3)$$

$$I\ e \rightarrow e$$

You would never want to program in this language—it doesn't even have variables!—but it's exactly as powerful as the  $\lambda$ -calculus.

# Bracket Abstraction

The function  $[x]$  that takes a combinator term  $M$  and builds another term that behaves like  $\lambda x.M$ :

$$\begin{aligned} [x] x &= I \\ [x] N &= K N && \text{where } x \notin fv(N) \\ [x] N_1 N_2 &= S ([x] N_1) ([x] N_2) \end{aligned}$$

The idea is that  $([x] M) N \rightarrow M\{N/x\}$  for every term  $N$ .

# Bracket Abstraction

We then define a function  $(e)^*$  that maps a  $\lambda$ -calculus expression to a combinator term:

$$\begin{aligned}(x)^* &= x \\ (e_1 e_2)^* &= (e_1)^* (e_2)^* \\ (\lambda x. e)^* &= [x] (e)^*\end{aligned}$$

# Example

As an example, the expression  $\lambda x. \lambda y. x$  is translated as follows:

$$\begin{aligned} & (\lambda x. \lambda y. x)^* \\ = & [x] (\lambda y. x)^* \\ = & [x] ([y] x) \\ = & [x] (K x) \\ = & (S ([x] K) ([x] x)) \\ = & S (K K) I \end{aligned}$$

No variables in the final combinator term!

# Example

---

We can check that this behaves the same as our original  $\lambda$ -expression by seeing how it evaluates when applied to arbitrary expressions  $e_1$  and  $e_2$ .

$$\begin{aligned} & (\lambda x. \lambda y. x) e_1 e_2 \\ = & (\lambda y. e_1) e_2 \\ = & e_1 \end{aligned}$$



# Example

We can check that this behaves the same as our original  $\lambda$ -expression by seeing how it evaluates when applied to arbitrary expressions  $e_1$  and  $e_2$ .

$$\begin{aligned} & (\lambda x. \lambda y. x) e_1 e_2 \\ = & (\lambda y. e_1) e_2 \\ = & e_1 \end{aligned}$$

and

$$\begin{aligned} & (S (K K) I) e_1 e_2 \\ = & (K K e_1) (I e_1) e_2 \\ = & K e_1 e_2 \\ = & e_1 \end{aligned}$$

# SKI Without I

Looking back at our definitions...

$$K e_1 e_2 \rightarrow e_1$$

$$S e_1 e_2 e_3 \rightarrow e_1 e_3 (e_2 e_3)$$

$$I e \rightarrow e$$

...I isn't strictly necessary. It equals S K K.

# SKI Without I

Looking back at our definitions...

$$K e_1 e_2 \rightarrow e_1$$

$$S e_1 e_2 e_3 \rightarrow e_1 e_3 (e_2 e_3)$$

$$I e \rightarrow e$$

...I isn't strictly necessary. It equals S K K.

Our example becomes:

$$S (K K) (S K K)$$

# Encodings

---

The pure  $\lambda$ -calculus contains only functions as values. It is not exactly easy to write large or interesting programs in the pure  $\lambda$ -calculus. We can however encode objects, such as booleans, and integers.

# Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

AND TRUE FALSE = FALSE

NOT FALSE = TRUE

IF TRUE  $e_1$   $e_2$  =  $e_1$

IF FALSE  $e_1$   $e_2$  =  $e_2$

# Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

AND TRUE FALSE = FALSE

NOT FALSE = TRUE

IF TRUE  $e_1$   $e_2$  =  $e_1$

IF FALSE  $e_1$   $e_2$  =  $e_2$

Let's start by defining TRUE and FALSE:

TRUE  $\triangleq$   $\lambda x. \lambda y. x$

FALSE  $\triangleq$   $\lambda x. \lambda y. y$

# Booleans

---

We want the function IF to behave like

$$\lambda b. \lambda t. \lambda f. \text{if } b = \text{TRUE} \text{ then } t \text{ else } f.$$

# Booleans

We want the function IF to behave like

$$\lambda b. \lambda t. \lambda f. \text{if } b = \text{TRUE} \text{ then } t \text{ else } f.$$

The definitions for TRUE and FALSE make this very easy.

$$\text{IF} \triangleq \lambda b. \lambda t. \lambda f. b \ t \ f$$



# Booleans

We want the function IF to behave like

$$\lambda b. \lambda t. \lambda f. \text{if } b = \text{TRUE} \text{ then } t \text{ else } f.$$

The definitions for TRUE and FALSE make this very easy.

$$\text{IF} \triangleq \lambda b. \lambda t. \lambda f. b \ t \ f$$

We can also write the standard Boolean operators.

$$\text{NOT} \triangleq \lambda b. b \ \text{FALSE} \ \text{TRUE}$$

$$\text{AND} \triangleq \lambda b_1. \lambda b_2. b_1 \ b_2 \ \text{FALSE}$$

$$\text{OR} \triangleq \lambda b_1. \lambda b_2. b_1 \ \text{TRUE} \ b_2$$

# Church Numerals

---

Let's encode the natural numbers!

We'll write  $\bar{n}$  for the encoding of the number  $n$ . The central function we'll need is a *successor* operation:

$$\text{SUCC } \bar{n} = \overline{n + 1}$$

# Church Numerals

Church numerals encode a number  $n$  as a function that takes  $f$  and  $x$ , and applies  $f$  to  $x$   $n$  times.

$$\begin{aligned}\bar{0} &\triangleq \lambda f. \lambda x. x \\ \bar{1} &\triangleq \lambda f. \lambda x. f x \\ \bar{2} &\triangleq \lambda f. \lambda x. f(f x)\end{aligned}$$

# Church Numerals

Church numerals encode a number  $n$  as a function that takes  $f$  and  $x$ , and applies  $f$  to  $x$   $n$  times.

$$\bar{0} \triangleq \lambda f. \lambda x. x$$

$$\bar{1} \triangleq \lambda f. \lambda x. f x$$

$$\bar{2} \triangleq \lambda f. \lambda x. f(f x)$$

This makes it easy to write the successor function:

$$\text{SUCC} \triangleq \lambda n. \lambda f. \lambda x. f(n f x)$$

# Addition

---

Given the definition of SUCC, we can define addition. Intuitively, the natural number  $n_1 + n_2$  is the result of applying the successor function  $n_1$  times to  $n_2$ .

$$\text{PLUS} \triangleq \lambda n_1. \lambda n_2. n_1 \text{ SUCC } n_2$$