CS 4110

Programming Languages & Logics

Lecture 15
De Bruijn, Combinators, Encodings
Review: \(\lambda\)-calculus

**Syntax**

\[
e \ ::= \ x \mid e_1 \ e_2 \mid \lambda x. \ e \\
\]

**Semantics**

\[
\begin{align*}
& e_1 \rightarrow e_1' \\
\hline
& e_1 \ e_2 \rightarrow e'_1 \ e_2 \\
& e \rightarrow e' \\
& v \ e \rightarrow v \ e' \\
\end{align*}
\]

\[
(\lambda x. \ e) \ v \rightarrow e\{v/x\}^\beta
\]
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.
Another way to avoid the tricky issues with substitution is to use a *nameless* representation of terms.

\[ n \in \text{Int} \]

\[ e ::= n \mid \lambda.e \mid e\;e \]

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

\[
\lambda. \lambda. 1 \quad \lambda. 1. \lambda. 2
\]

\[
\lambda z. \lambda x. \lambda y. x \quad \lambda. \lambda. \lambda. 1
\]
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:

<table>
<thead>
<tr>
<th>Standard</th>
<th>de Bruijn</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x. x$</td>
<td>$\lambda. 0$</td>
</tr>
<tr>
<td>$\lambda z. z$</td>
<td>$\lambda. 0$</td>
</tr>
</tbody>
</table>
Here are some terms written in standard and de Bruijn notation:

<table>
<thead>
<tr>
<th>Standard</th>
<th>de Bruijn</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x. x$</td>
<td>$\lambda. 0$</td>
</tr>
<tr>
<td>$\lambda z. z$</td>
<td>$\lambda. 0$</td>
</tr>
<tr>
<td>$\lambda x. \lambda y. x$</td>
<td>$\lambda. 0$</td>
</tr>
</tbody>
</table>
Examples

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

<table>
<thead>
<tr>
<th>Standard</th>
<th>de Bruijn</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda x. x)</td>
<td>(\lambda. 0)</td>
</tr>
<tr>
<td>(\lambda z. z)</td>
<td>(\lambda. 0)</td>
</tr>
<tr>
<td>(\lambda x. \lambda y. x)</td>
<td>(\lambda. \lambda. 1)</td>
</tr>
<tr>
<td>(\lambda x. \lambda y. \lambda s. \lambda z. x s (y s z))</td>
<td></td>
</tr>
</tbody>
</table>
Examples

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

<table>
<thead>
<tr>
<th>Standard</th>
<th>de Bruijn</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x. \ x )</td>
<td>( \lambda. \ 0 )</td>
</tr>
<tr>
<td>( \lambda z. \ z )</td>
<td>( \lambda. \ 0 )</td>
</tr>
<tr>
<td>( \lambda x. \ \lambda y. \ x )</td>
<td>( \lambda. \ \lambda. \ 1 )</td>
</tr>
<tr>
<td>( \lambda x. \ \lambda y. \ \lambda s. \ \lambda z. \ x \ s \ (y \ s \ z) )</td>
<td>( \lambda. \ \lambda. \ \lambda. \ \lambda. \ 3 \ 1 \ (2 \ 1 \ 0) )</td>
</tr>
<tr>
<td>( \lambda x. \ x \ x ) \ (\lambda x. \ x \ x) )</td>
<td></td>
</tr>
</tbody>
</table>
Here are some terms written in standard and de Bruijn notation:

<table>
<thead>
<tr>
<th>Standard</th>
<th>de Bruijn</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x. x )</td>
<td>( \lambda. 0 )</td>
</tr>
<tr>
<td>( \lambda z. z )</td>
<td>( \lambda. 0 )</td>
</tr>
<tr>
<td>( \lambda x. \lambda y. x )</td>
<td>( \lambda. \lambda. 1 )</td>
</tr>
<tr>
<td>( \lambda x. \lambda y. \lambda s. \lambda z. x s (y s z) )</td>
<td>( \lambda. \lambda. \lambda. \lambda. 3 \ 1 \ (2 \ 1 \ 0) )</td>
</tr>
<tr>
<td>((\lambda x. x x) \ (\lambda x. x x))</td>
<td>((\lambda. 0 \ 0) \ (\lambda. 0 \ 0))</td>
</tr>
<tr>
<td>((\lambda x. \lambda x. x) \ (\lambda y. y))</td>
<td></td>
</tr>
</tbody>
</table>
Here are some terms written in standard and de Bruijn notation:

<table>
<thead>
<tr>
<th>Standard</th>
<th>de Bruijn</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x. x$</td>
<td>$\lambda. 0$</td>
</tr>
<tr>
<td>$\lambda z. z$</td>
<td>$\lambda. 0$</td>
</tr>
<tr>
<td>$\lambda x. \lambda y. x$</td>
<td>$\lambda. \lambda. 1$</td>
</tr>
<tr>
<td>$\lambda x. \lambda y. \lambda s. \lambda z. x s (y s z)$</td>
<td>$\lambda. \lambda. \lambda. \lambda. 31 (210)$</td>
</tr>
<tr>
<td>$(\lambda x. x x) (\lambda x. x x)$</td>
<td>$(\lambda. 00) (\lambda. 00)$</td>
</tr>
<tr>
<td>$(\lambda x. \lambda x. x) (\lambda y. y)$</td>
<td>$(\lambda. \lambda. 0) (\lambda. 0)$</td>
</tr>
</tbody>
</table>
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 $\lambda z. (x y) z$ is 1 2 0
Shifting

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

\[
\uparrow^i_c (n) = \begin{cases} 
  n & \text{if } n < c \\
  n + i & \text{otherwise}
\end{cases}
\]

\[
\uparrow^i_c (\lambda.e) = \lambda.(\uparrow^i_{c+1} e)
\]

\[
\uparrow^i_c (e_1 e_2) = (\uparrow^i_c e_1)(\uparrow^i_c e_2)
\]

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 as follows:

\[
\begin{align*}
    n\{e/m\} &= \begin{cases} 
        e & \text{if } n = m \\
        n & \text{otherwise}
    \end{cases} \\
    (\lambda.e_1)\{e/m\} &= \lambda.e_1\{(\uparrow_0^1 e)/m + 1\}) \\
    (e_1 \ e_2)\{e/m\} &= (e_1\{e/m\}) \ (e_2\{e/m\})
\end{align*}
\]
Substitution

Now we can define substitution as follows:

$$n\{e/m\} = \begin{cases} e & \text{if } n = m \\ n & \text{otherwise} \end{cases}$$

$$\lambda.e_1\{e/m\} = \lambda.e_1\{(\uparrow_0^1 e)/m + 1\})$$

$$e_1\;e_2\{e/m\} = (e_1\{e/m\})\;(e_1\{e/m\})$$

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

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

$$(\lambda x.\;e_1)\;e_2 \rightarrow e_c\{e_2/x\}$$
Example

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

\[
(\lambda. \, \lambda. \, 12) \, 1
\]
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 \(\Gamma(y) = 1\).

\[
(\lambda. \lambda. 1\ 2)\ 1 \\
\rightarrow \uparrow_{0}^{-1} (((\lambda. 1\ 2)\{(\uparrow_{0}^{1}\ 1)/0\}))
\]

\[
\rightarrow 2
\]
Example

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

\[
(\lambda. \lambda. 1 \, 2) \, 1 \\
\Rightarrow \uparrow_0^{-1} (((\lambda. 1 \, 2)\{((\uparrow_0^{1} 1)/0}\}) \\
= \uparrow_0^{-1} (((\lambda. 1 \, 2)\{2/0}\})
\]
Consider the term \((\lambda u. \lambda v. u \ x) \ y\) with respect to a context where \(\Gamma(x) = 0\) and \(\Gamma(y) = 1\).

\[
\begin{align*}
(\lambda.\lambda.1\ 2)\ 1 & \rightarrow \uparrow_{0}^{-1} ((\lambda.1\ 2)\{(\uparrow_{0}^{1} 1)/0\}) \\
& = \uparrow_{0}^{-1} ((\lambda.1\ 2)\{2/0\}) \\
& = \uparrow_{0}^{-1} \lambda.((1\ 2)\{(\uparrow_{0}^{1} 2)/(0 + 1)\})
\end{align*}
\]
Example

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

\[
(\lambda. \lambda.1 \ 2) \ 1 \\
\rightarrow \uparrow^{-1}_0 (((\lambda.1 \ 2)\{((\uparrow^1_0 1)/0\})) \\
= \uparrow^{-1}_0 (((\lambda.1 \ 2)\{2/0\})) \\
= \uparrow^{-1}_0 \lambda.((1 \ 2)\{((\uparrow^1_0 2)/(0 + 1))\}) \\
= \uparrow^{-1}_0 \lambda.((1 \ 2)\{3/1\})
\]
Example

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

\[
(\lambda. \lambda. 1 \ 2) \ 1 \\
\rightarrow \uparrow^{-1}_0 (((\lambda. 1 \ 2)\{((\uparrow^{1}_0 \ 1)/0\}) \\
= \uparrow^{-1}_0 ((\lambda. 1 \ 2)\{2/0\}) \\
= \uparrow^{-1}_0 \lambda.((1 \ 2)\{((\uparrow^{1}_0 \ 2)/(0 + 1))\}) \\
= \uparrow^{-1}_0 \lambda.((1 \ 2)\{3/1\}) \\
= \uparrow^{-1}_0 \lambda.(1\{3/1\}) \ (2\{3/1\})
\]
Consider the term \((\lambda u.\lambda v.u\ x)\ y\) with respect to a context where \(\Gamma(x) = 0\) and \(\Gamma(y) = 1\).

\[
(\lambda.\lambda.1\ 2)\ 1
\]

\[
\rightarrow \uparrow_{0}^{-1} ((\lambda.1\ 2)\{\uparrow_{0}^{1} 1/0\})
\]

\[
= \uparrow_{0}^{-1} ((\lambda.1\ 2)\{2/0\})
\]

\[
= \uparrow_{0}^{-1} \lambda.((1\ 2)\{\uparrow_{0}^{1} 2/(0 + 1)\})
\]

\[
= \uparrow_{0}^{-1} \lambda.((1\ 2)\{3/1\})
\]

\[
= \uparrow_{0}^{-1} \lambda.(1\{3/1\})\ (2\{3/1\})
\]

\[
= \uparrow_{0}^{-1} \lambda.3\ 2
\]
Example

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

\[
(\lambda.\lambda.1\ 2) \ 1
\]

\[
\rightarrow \uparrow_{0}^{-1} (((\lambda.1\ 2)\{(\uparrow_{0}^{1}\ 1)/0\}))
\]

\[
= \uparrow_{0}^{-1} (((\lambda.1\ 2)\{2/0\})
\]

\[
= \uparrow_{0}^{-1} \lambda.((1\ 2)\{(\uparrow_{0}^{1}\ 2)/(0 + 1)\})
\]

\[
= \uparrow_{0}^{-1} \lambda.((1\ 2)\{3/1\})
\]

\[
= \uparrow_{0}^{-1} \lambda.(1\{3/1\})\ (2\{3/1\})
\]

\[
= \uparrow_{0}^{-1} \lambda.3 \ 2
\]

\[
= \lambda.2 \ 1
\]

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

\[
(\lambda. \lambda. 1 \ 2) \ 1 \\
\rightarrow \ \uparrow_0^- (((\lambda. 1 \ 2)\left((\uparrow_0^1 1)/0\right)) \\
= \ \uparrow_0^- (((\lambda. 1 \ 2)\{2/0\}) \\
= \ \uparrow_0^- \ \lambda.((1 \ 2)\left((\uparrow_0^1 2)/(0 + 1)\right)) \\
= \ \uparrow_0^- \ \lambda.((1 \ 2)\{3/1\}) \\
= \ \uparrow_0^- \ \lambda.((1\{3/1\}) \ (2\{3/1\})) \\
= \ \uparrow_0^- \ \lambda.3 \ 2 \\
= \ \lambda.2 \ 1
\]

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 λ-calculus is to work with closed expressions or *combinators*.

With just three combinators, we can encode the entire λ-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 \textit{combinators}.

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

\[
\begin{align*}
K &= \lambda x. \lambda y. x \\
S &= \lambda x. \lambda y. \lambda z. x \ z \ (y \ z) \\
I &= \lambda x. x
\end{align*}
\]
Combinators

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

Behold the “SKI-calculus”:

\[
\begin{align*}
K & \ e_1 \ e_2 \rightarrow e_1 \\
S & \ e_1 \ e_2 \ e_3 \rightarrow e_1 \ e_3 \ (e_2 \ e_3) \\
I & \ e \rightarrow e
\end{align*}
\]

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 \(\lambda x. M\):

\[
\begin{align*}
[x] x & = I \\
[x] N & = KN \\
[x] N_1 N_2 & = S ([x] N_1) ([x] N_2)
\end{align*}
\]

where \(x \notin \text{fv}(N)\)

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:

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

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

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

$= [x] (\lambda y. x)^*$$

$= [x] ([y] x)$$

$= [x] (K x)$$

$= (S ([x] K) ([x] x))$$

$= S (K K) I$$

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 $e_2$.

\[
(\lambda x. \lambda y. x) \, e_1 \, e_2 \\
= (\lambda y. \, e_1) \, e_2 \\
= e_1
\]
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 \( e_2 \).

\[
\begin{align*}
(\lambda x. \lambda y. x) \; e_1 \; e_2 &= (\lambda y. e_1) \; e_2 \\
&= e_1
\end{align*}
\]

and

\[
\begin{align*}
(S \; (K \; K) \; I) \; e_1 \; e_2 &= (K \; K \; e_1) \; (I \; e_1) \; e_2 \\
&= K \; e_1 \; e_2 \\
&= e_1
\end{align*}
\]
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.
Looking back at our definitions...

\[
\begin{align*}
  K e_1 e_2 & \rightarrow e_1 \\
  S e_1 e_2 e_3 & \rightarrow e_1 e_3 (e_2 e_3) \\
  I e & \rightarrow e
\end{align*}
\]

...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:

\[
\begin{align*}
\text{AND}&\quad \text{TRUE FALSE} = \text{FALSE} \\
\text{NOT}&\quad \text{FALSE} = \text{TRUE} \\
\text{IF }\quad \text{TRUE } e_1 e_2 &= e_1 \\
\text{IF }\quad \text{FALSE } e_1 e_2 &= e_2
\end{align*}
\]
Booleans

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

\[
\begin{align*}
\text{AND TRUE FALSE} &= \text{FALSE} \\
\text{NOT FALSE} &= \text{TRUE} \\
\text{IF TRUE } e_1 e_2 &= e_1 \\
\text{IF FALSE } e_1 e_2 &= e_2
\end{align*}
\]

Let’s start by defining TRUE and FALSE:

\[
\begin{align*}
\text{TRUE} & \triangleq \lambda x. \lambda y. x \\
\text{FALSE} & \triangleq \lambda x. \lambda y. y
\end{align*}
\]
Booleans

We want the function IF to behave like

\[ \lambda b. \lambda t. \lambda f. \text{if } b = \text{TRUE then } t \text{ else } f. \]
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:

$$\text{SUCC } \bar{n} = n + 1$$
Church Numerals

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

\[
\begin{align*}
0 & \triangleq \lambda f. \lambda x. x \\
1 & \triangleq \lambda f. \lambda x. f x \\
2 & \triangleq \lambda f. \lambda x. f (f x)
\end{align*}
\]
Church Numerals

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

\[
\begin{align*}
\bar{0} & \triangleq \lambda f. \lambda x. x \\
\bar{1} & \triangleq \lambda f. \lambda x. f x \\
\bar{2} & \triangleq \lambda f. \lambda x. f (f x)
\end{align*}
\]

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 $n_2$.

\[
\text{PLUS} \triangleq \lambda n_1. \lambda n_2. n_1 \text{ SUCC } n_2
\]