Review: $\lambda$-calculus

Syntax

$$e ::= x \mid e_1 e_2 \mid \lambda x. e$$

$$v ::= \lambda x. e$$

Semantics

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

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

<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></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></td>
</tr>
</tbody>
</table>
Here are some terms written in standard and de Bruijn notation:

<table>
<thead>
<tr>
<th><strong>Standard</strong></th>
<th><strong>de Bruijn</strong></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>
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></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>( (\lambda. 0 \ 0) \ (\lambda. 0 \ 0) )</td>
</tr>
<tr>
<td>( (\lambda x. \lambda x. x) (\lambda y. y) )</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>((\lambda. 0 0) (\lambda. 0 0))</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 $01$
- Representation of $\lambda z. xyz \, \lambda. 120$
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:

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

\[ (\lambda.\ e_1)\{e/m\} = \lambda.\ e_1\{(\uparrow^1_0\ e)/m + 1\} \]

\[ (e_1\ e_2)\{e/m\} = (e_1\{e/m\})\ (e_2\{e/m\}) \]
Substitution

Now we can define substitution:

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

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

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

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

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

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

\[(\lambda.\lambda.1 \, 2) \, 1\]
Consider the term \((\lambda u.\lambda v. u\ x)\ y\) with respect to a context where \(\Gamma(x) = 0\) and \(\Gamma(y) = 1\).

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

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

\[
(\lambda.\lambda.1\ 2) \ 1 \\
\rightarrow \uparrow_0^{-1} \ ((\lambda.1\ 2)\{ (\uparrow_0^1 1)/0 \}) \\
= \uparrow_0^{-1} \ ((\lambda.1\ 2)\{2/0\})
\]
Example

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

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

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

Example

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

\[
\begin{align*}
(\lambda.\lambda.1 2) 1 \\
\rightarrow \uparrow_{\emptyset}^{\perp} ((\lambda.1 2)\{\uparrow_{\emptyset}^{1} 1/0\}) \\
= \uparrow_{\emptyset}^{\perp} ((\lambda.1 2)\{2/0\}) \\
= \uparrow_{\emptyset}^{\perp} \lambda.((1 2)\{\uparrow_{\emptyset}^{1} 2/(0 + 1)\}) \\
= \uparrow_{\emptyset}^{\perp} \lambda.((1 2)\{3/1\})
\end{align*}
\]
Example

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

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

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

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

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

\[
\begin{align*}
(\lambda \cdot \lambda.12) & \; 1 \\
\rightarrow & \; \uparrow_0^{-1} \left( (\lambda.12) \{ (\uparrow_0^1 1)/0 \} \right) \\
= & \; \uparrow_0^{-1} \left( (\lambda.12) \{ 2/0 \} \right) \\
= & \; \uparrow_0^{-1} \lambda.((12) \{ (\uparrow_0^1 2)/(0+1) \}) \\
= & \; \uparrow_0^{-1} \lambda.((12) \{ 3/1 \}) \\
= & \; \uparrow_0^{-1} \lambda.((1\{3/1\}) \cdot (2\{3/1\})) \\
= & \; \uparrow_0^{-1} \lambda.3 \; 2 \\
= & \; \lambda.2 \; 1
\end{align*}
\]
Consider the term \((\lambda u. \lambda v. u \ x) \ y\) with respect to a context where \(\Gamma(x) = 0\) and \(\Gamma(y) = 1\).

\[
(\lambda. \lambda. 1 \ 2) \ 1 \\
\rightarrow \uparrow_0^{-1} (((\lambda.1 \ 2)\{(\uparrow^1_0 \ 1)/0\})) \\
= \uparrow_0^{-1} (((\lambda.1 \ 2)\{2/0\})) \\
= \uparrow_0^{-1} \lambda.((1 \ 2)\{(\uparrow^1_0 \ 2)/(0 + 1)\})) \\
= \uparrow_0^{-1} \lambda.((1 \ 2)\{3/1\}) \\
= \uparrow_0^{-1} \lambda.(1\{3/1\}) \ (2\{3/1\}) \\
= \uparrow_0^{-1} \lambda.3 \ 2 \\
= \lambda.2 \ 1
\]

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

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

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 \(\lambda x. M\):

\[
\begin{align*}
[\mathit{x}] \mathit{x} & = \mathit{I} \\
[\mathit{x}] \mathit{N} & = \mathit{K} \mathit{N} \\
[\mathit{x}] \mathit{N}_1 \mathit{N}_2 & = \mathit{S} ([\mathit{x}] \mathit{N}_1) ([\mathit{x}] \mathit{N}_2)
\end{align*}
\]

where \(x \not\in \text{fv}(\mathit{N})\)

The idea is that \(([x] M) \mathit{N} \rightarrow M\{\mathit{N}/x\}\) for every term \(\mathit{N}\).
We then define a function \((e)\ast\) that maps a \(\lambda\)-calculus expression to a combinator term:

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

$$(\lambda x. \lambda y. x) e_1 e_2$$

$\rightarrow (\lambda y. e_1) e_2$

$\rightarrow e_1$$
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 \( e_2 \).

\[
(\lambda x. \lambda y. x) \ e_1 \ e_2 \\
\rightarrow (\lambda y. e_1) \ e_2 \\
\rightarrow e_1
\]

and

\[
(S \ (K \ K) \ I) \ e_1 \ e_2 \\
\rightarrow (K \ K \ e_1) \ (I \ e_1) \ e_2 \\
\rightarrow K \ e_1 \ e_2 \\
\rightarrow e_1
\]
Looking back at our definitions...

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

.../ isn’t strictly necessary. It behaves the same as S K K.
Looking back at our definitions...

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

...I isn’t strictly necessary. It behaves the same as 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:

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

\[
\begin{align*}
\text{TRUE} & \triangleq \\
\text{FALSE} & \triangleq
\end{align*}
\]
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:

\[
\begin{align*}
\text{TRUE} & \triangleq \lambda x. \lambda y. x \\
\text{FALSE} & \triangleq \lambda x. \lambda y. y
\end{align*}
\]
We want the function IF to behave like

\[ \lambda b. \lambda t. \lambda f. \text{if } b \text{ is our term TRUE then } t, \text{ otherwise } f \]
Booleans

We want the function IF to behave like

\[ \lambda b. \lambda t. \lambda f. \text{if } b \text{ is our term TRUE then } t, \text{ otherwise } f \]

We can rely on the way we defined TRUE and FALSE:

\[ \text{IF } \triangleq \lambda b. \lambda t. \lambda f. b \, t \, f \]
Booleans

We want the function IF to behave like

$$\lambda b. \lambda t. \lambda f. \text{if } b \text{ is our term TRUE then } t, \text{ otherwise } f$$

We can rely on the way we defined TRUE and FALSE:

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

We can also write the standard Boolean operators.

$$\text{NOT } \triangleq \quad \text{AND } \triangleq \quad \text{OR } \triangleq$$
Booleans

We want the function IF to behave like

\[ \lambda b. \lambda t. \lambda f. \text{if } b \text{ is our term TRUE then } t, \text{ otherwise } f \]

We can rely on the way we defined TRUE and FALSE:

\[ \text{IF} \triangleq \lambda b. \lambda t. \lambda f. b \, t \, f \]

We can also write the standard Boolean operators.

\[ \text{NOT} \triangleq \lambda b. \ b \ \text{FALSE} \ \text{TRUE} \]
\[ \text{AND} \triangleq \lambda b_1. \lambda b_2. \ b_1 \ b_2 \ \text{FALSE} \]
\[ \text{OR} \triangleq \lambda b_1. \lambda b_2. \ b_1 \ \text{TRUE} \ b_2 \]
Let’s encode the natural numbers!

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

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

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

$$\bar{0} \triangleq \lambda f. \lambda x. x$$
$$\bar{1} \triangleq \lambda f. \lambda x. f x$$
$$\bar{2} \triangleq \lambda f. \lambda x. f (f x)$$
Church Numerals

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

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

We can write a successor function that “inserts” another application of \( f \):

\[
\text{SUCC} \triangleq \lambda n. \lambda f. \lambda x. f (n f x)
\]
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 $n_2$.

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

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