

---

CMCS 312: Programming Languages

Lecture 6: de Bruijn Indices

Umut A. Acar

October 20, 2005

---

## Contents

<b>1</b>	<b>Announcements</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>1</b>
2.1	de Bruijn Indices . . . . .	1
2.2	Shifting and Substitution . . . . .	3

## 1 Announcements

Homework 2 is out today. It is due next Tuesday (Oct 17) before class. The homework includes both programming and pen-and-pencil problems. Please e-mail your programs to the TA. For the pen-and-pencil part, either e-mail your solutions, or hand them in at the beginning of the class.

This assignment can take you a few days, so start early.

## 2 Introduction

Last class we finished the discussion of untyped lambda calculus. Defining the calculus was easy. Programming with lambdas can be a bit tricky but it was fun. We even showed that we can write recursive function without having the ability to name functions themselves.

In this class, we will talk about de Bruijn indices. This is a neat trick that allows us to do substitution without doing lambda conversions.

### 2.1 de Bruijn Indices

Lambda conversions require coming up with fresh names on the fly. This can be quite difficult, so people come up with ideas to get around this problem. In this class, we will talk about a particular idea due to de Bruijn (pronounced “de brown”).

Consider the term  $(\lambda x. \lambda y. x y) (\lambda x.x)$  and its abstract syntax tree (see Figure 1).

Can we represent the same AST without using variable names? Yes, the variable names do nothing but refer to the lambda where that variable is bound. So we can simply represent them explicitly via pointers.

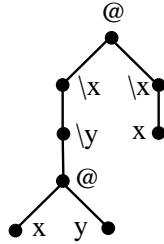


Figure 1: The ASTs for  $(\lambda x. \lambda y. x y) (\lambda x. x)$  with pointers.

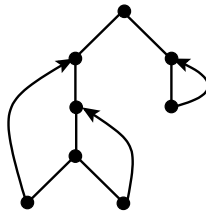


Figure 2: The ASTs for  $(\lambda x. \lambda y. x y) (\lambda x.x)$  with pointers.

Figure 2 shows such a representation. Note that when drawing the ASTs for lambda term with pointers, the labels are redundant. The degree of a node determines the kind of a term (application has degree two, lambda abstraction has degree one, and the leaves are variables). This is the key idea behind de Bruijn indices.

It is going to be difficult to work with AST's. Can we find a textual representation for de Bruijn's AST's?

How about representing pointers as the distance they travel? For example, the AST in Figure 2 can be written as  $(\lambda. \lambda. 1 0) (\lambda. 0)$ . If given the textual representation based on distances, we can construct the AST, then this approach is sound. But is it?

The answer is yes. The reason is that the pointers can only go from one node to a node that is on the path from that node to the root. So distances, uniquely identify the binding location. But why is that the pointers can only go up along the path from the node to the root? Because a node can only refer to a variable bound by a lambda above it. So the representation based on the distances works. The distances are known as *de Bruijn* indices.

Let's develop the idea that a node can refer only to a variable that is bound on the path to the root. Let's denote the (*naming*) context, denoted  $\Gamma$  of a node (*i.e.*, a term) as a relation that maps variables to their de Bruijn indices. We can write  $\Gamma$  as a sequence  $\Gamma = x_n x_{n-1} \dots x_0$  to represent the relation  $\{(x_n, n), (x_{n-1}, n-1), \dots, (x_0, 0)\}$ . In other words, the de Bruijn index of a variable is determined by its position in the sequence. Note that there can be multiple instance of the same variable, *e.g.*  $\Gamma = xyzxy$ .

How can we construct the naming context for each node in an AST?

Consider some node in the tree and suppose we have the current context,  $\Gamma$ , for that node. We can construct the context for its subtree(s) as follows. If the root node is not a lambda then the same context is passed to the subtree(s). If the root is a lambda then we pass  $\Gamma, x$  to the subtrees. This extends  $\Gamma$  with  $x$ , which is given an index of 0, and it shifts the indices of all other variables by one, which is required because they are now further away. To construct the naming context for each node, we apply this idea starting at the root node. Note that the subtrees can be traversed in parallel, so we need not specify the ordering.

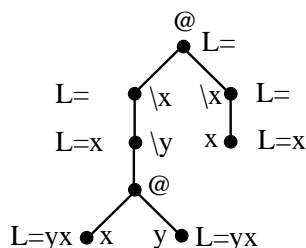


Figure 3: The ASTs for  $(\lambda x. \lambda y. x y) (\lambda x. x)$  with naming contexts.

For example, Figure 3 shows the naming context for each node (or term) in the term  $(\lambda x. \lambda y. x y) (\lambda x. x)$ .

Using the naming contexts, we can transform a term into its de Bruijn representation. This is relatively straightforward. Just replace each variable node with the smallest index of the variable in the context.

Now that we know what is going on, let's write it out. Let  $dB(\Gamma, \cdot)$  be a function that transforms a lambda term to a de-Bruijn term under the context lambda.

$$\begin{aligned}
 dB(\Gamma, x) &= \Gamma_x \\
 dB(\Gamma, \lambda x. t) &= \lambda. dB((\Gamma, x), t) \\
 dB(t_1 t_2) &= dB(t_1) dB(t_2)
 \end{aligned}$$

## 2.2 Shifting and Substitution

We can visualize beta reduction as an operation on AST's as substituting the right subtree of a degree two node into the left subtree at the leaves specified by the variable name being bound. For example, Figure 4 show a substitution. Since the term being substituted has not free variables, we need not work about alpha conversion.

How does substitution work with de Bruijn indices? It is very similar, except that we don't need to worry about renaming whatsoever. It just works. This example is somewhat simple because the term being substituted term does not have any free variables.

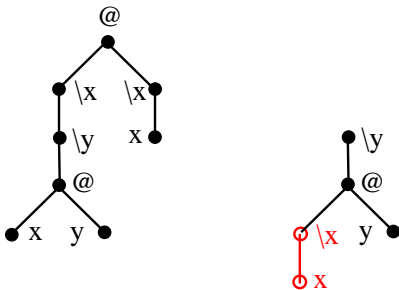


Figure 4:  $(\lambda x. \lambda y. x y) (\lambda x. x) \rightarrow_{\beta} \lambda y. (\lambda x. x) y$  illustrated.

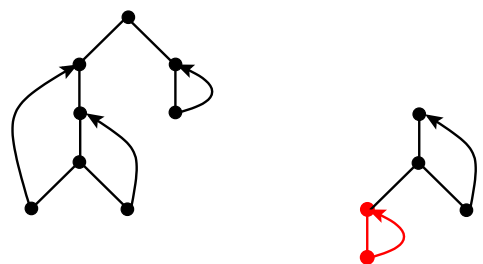


Figure 5:  $(\lambda \lambda. \lambda. 1 0) (\lambda. 0) \rightarrow_{\beta} \lambda (\lambda. 0) 0$  illustrated

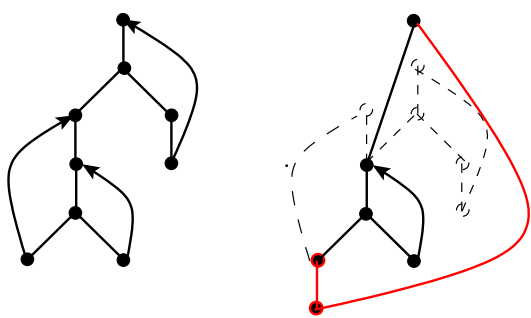


Figure 6:  $(\lambda \lambda. \lambda. \lambda. 10) (\lambda. 1) \rightarrow_{\beta} \lambda (\lambda \lambda. 2 0)$  illustrated

What happens with free variables? Nothing interesting, they just keep pointing to the same node that they used to (no binding location that the subtree being substituted can disappear). Figure 6 shows an example. The deleted elements are shown dashed.

Let's write out a formula for substitution. To get some intuition, consider the last example. The reduction,  $(\lambda.(\lambda.\lambda.1\ 0)(\lambda.1)) \rightarrow_{\beta} \lambda(\lambda\lambda.2\ 0)$ , transforms an expression into something totally unrecognizable. Let's see what is going on.

Imagine yourself substituting a tree  $T$  in place another some leaf node. Remember that during substitution the pointers keep pointing to the same binding locations. But then their length may change—we may have to stretch them. Which pointers stretch?

Only the pointers that point outside  $T$  (free variables) stretch. Let  $t$  be the term for  $T$ , to reflect the stretching, we will have to shift  $t$ . Let's define a  $d$ -place shift of a term  $t$ , denoted  $\uparrow_c^d(t)$ , as a term where all the free terms are shifted by  $d$  (increased by  $d$ ). But how do we identify which variables are free? The key insight is that the bound variables always constitute the smallest indices in a term (because they are closest). We can thus carry along a parameter called *cutoff* that is set the smallest index a free variable may have. Here is the definition:

$$\begin{aligned}\uparrow_c^d(i) &= i && \text{if } i < c \\ \uparrow_c^d(i) &= i + d && \text{if } i \geq c \\ \uparrow_c^d(\lambda.t) &= \lambda.\uparrow_d^{c+1}(t) \\ \uparrow_c^d(t_1\ t_2) &= (\uparrow_c^d(t_1))\ (\uparrow_c^d(t_2))\end{aligned}$$

Examples:

1.  $\uparrow^2(\lambda.\lambda.1(0\ 2)) =$
2.  $\uparrow^2(\lambda.01(\lambda.0\ 1\ 2)) =$

We can now define substitution.

$$\begin{aligned}[t/j]i &= t && \text{if } i = j \\ [t/j]i &= i && \text{if } i \neq j \\ [t/j]\lambda.t' &= \lambda.[\uparrow^1(t)/j + 1]t' \\ [t/j]t_1\ t_2 &= [t/j]t_1\ [t/j]t_2\end{aligned}$$

Based on substitution, we define beta reduction as

$$(\lambda.t_1)t_2 = \uparrow^{-1}([\uparrow^1(t_2)/0]t_1)$$