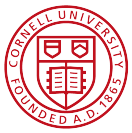
CS 4110 – Programming Languages and Logics
Lecture #15: De Bruijn, Combinators, Encodings



1 de Bruijn Notation

One way to avoid the tricky interaction between free and bound names in the substitution operator is to pick a representation for expressions that doesn't have any names at all! Intuitively, we can think of a bound variable is just a pointer to the λ that binds it. For example, in $\lambda x.\lambda y.y\ x$, the y points to the first λ and the x points to the second λ.

So-called *de Bruijn* notation uses this idea as the representation for λ expressions. Here is the grammar for λ expressions in de Bruijn notation:

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

Variables are represented by integers n that refer to (the index of) their binder while $lambda$-abstractions have the form $\lambda.e$. Note that the the variable bound by the abstraction is not named—i.e., the representation is nameless.

As examples, here are several terms written using standard notation and in de Bruijn notation:

Standard	de Bruijn
$\lambda x.x$	$\lambda.0$
$\lambda z.z$	$\lambda.0$
$\lambda x.\lambda y.x$	$\lambda.\lambda.1$
$\lambda x.\lambda y.\lambda s.\lambda z.x\ s\ (y\ s\ z)$	$\lambda.\lambda.\lambda.\lambda.3\ 1\ (2\ 1\ 0)$
$(\lambda x.x\ x)\ (\lambda x.x\ x)$	$(\lambda.0\ 0)\ (\lambda.0\ 0)$
$(\lambda x.\lambda x.x)\ (\lambda y.y)$	$(\lambda.\lambda.0)\ (\lambda.0)$

To represent a λ-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 Γ from variables to integers called a *context*. As an example, if Γ maps x to 0 and y to 1, then the de Bruijn representation of $x\ y$ with respect to Γ is $0\ 1$, while the representation of $\lambda z.\ x\ y\ z$ with respect to Γ is $\lambda.\ 1\ 2\ 0$. Note that in this second example, because we have gone under a λ, we have shifted the integers representing x and y up by one to avoid capturing them.

In general, whenever we work de Bruijn representations of expressions containing free variables (i.e., when working with respect to a context Γ) we will need to modify the indices of those variables. For example, when we substitute an expression containing free variables under a λ, we will need to shift the indices up so that they continue to refer to the same numbers with respect to Γ after the substitution as they did before. For example, if we substitute $0\ 1$ for the variable bound by the outermost λ in $\lambda.\lambda.1$ we should get $\lambda.\lambda.2\ 3$, not $\lambda.\lambda.0\ 1$. We will use an auxiliary function that shifts

the indices of free variables above a cutoff c up by i:

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

The cutoff keeps track of the variables that were bound in the original expression and so should not be shifted as the shifting operator walks down the structure of an expression. The cutoff is 0 initially.

Using this shifting function, we can define substitution as follows:

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

Note that when we go under a λ we increase the index of the variable we are substituting for and shift the free variables in the expression e up by one.

The β rule for terms in de Bruijn notation is as follows:

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

That is, we substitute occurrences of 0, the index of the variable being bound by the λ, with e_2 shifted up by one. Then we shift the result down by one to ensure that any free variables in e_1 continue to refer to the same things after we remove the λ.

To illustrate how this works consider the following example, which we wrote as $(\lambda u.\lambda v.u\ x)\ y$ in standard notation. We will work with respect to a context where $\Gamma(x) = 0$ and $\Gamma(y) = 1$.

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

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

2 Combinators

Yet 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*. It turns out that just using two combinators, S, K, and application, we can encode the entire λ-calculus.

Here are the evaluation rules for S, K, as well as a third combinator I, which will also be useful:

$$\begin{aligned}
&\mathsf{K}\ x\ y \to x \\
&\mathsf{S}\ x\ y\ z \to x\ z\ (y\ z) \\
&\mathsf{I}\ x \to x
\end{aligned}$$

Equivalently, here are their definitions as closed λ-expressions:

$$\begin{aligned}
\mathsf{K} &= \lambda x.\lambda y.\ x \\
\mathsf{S} &= \lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z) \\
\mathsf{I} &= \lambda x.\ x
\end{aligned}$$

It is not hard to see that I is not needed—it can be encoded as S K K.

To show how these combinators can be used to encode the λ-calculus, we have to define a translation that takes an arbitrary closed λ-calculus expression and turns it into a combinator term that behaves the same during evaluation. This translation is called *bracket abstraction*. It proceeds in two steps. First, we define a function $[x]$ that takes a combinator term M possibly containing free variables and builds another term that behaves like $\lambda x.M$, in the sense that $([x]\ M)\ N \to M\{N/x\}$ for every term N:

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

Second, we define a function $(e)*$ that maps a λ-calculus expression to a combinator term:

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

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

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

We can check that this behaves the same as our original λ-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

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

3 λ-calculus encodings

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

3.1 Booleans

Let us start by encoding constants and operators for booleans. That is, we want to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as expected. For example:

$$\begin{aligned} \mathsf{AND}\ \mathsf{TRUE}\ \mathsf{FALSE} &= \mathsf{FALSE} \\ \mathsf{NOT}\ \mathsf{FALSE} &= \mathsf{TRUE} \\ \mathsf{IF}\ \mathsf{TRUE}\ e_1\ e_2 &= e_1 \\ \mathsf{IF}\ \mathsf{FALSE}\ e_1\ e_2 &= e_2 \end{aligned}$$

Let's start by defining TRUE and FALSE:

$$\begin{aligned} \mathsf{TRUE} &\triangleq \lambda x.\, \lambda y.\, x \\ \mathsf{FALSE} &\triangleq \lambda x.\, \lambda y.\, y \end{aligned}$$

Thus, both TRUE and FALSE are functions that take two arguments; TRUE returns the first, and FALSE returns the second. We want the function IF to behave like

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

The definitions for TRUE and FALSE make this very easy.

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

Definitions of other operators are also straightforward.

$$\begin{aligned} \mathsf{NOT} &\triangleq \lambda b.\, b\ \mathsf{FALSE}\ \mathsf{TRUE} \\ \mathsf{AND} &\triangleq \lambda b_1.\, \lambda b_2.\, b_1\ b_2\ \mathsf{FALSE} \\ \mathsf{OR} &\triangleq \lambda b_1.\, \lambda b_2.\, b_1\ \mathsf{TRUE}\ b_2 \end{aligned}$$

3.2 Church numerals

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

$$\begin{aligned} \overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\ \overline{1} &= \lambda f.\, \lambda x.\, f\ x \\ \overline{2} &= \lambda f.\, \lambda x.\, f\ (f\ x) \\ \mathsf{SUCC} &\triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\ (n\ f\ x) \end{aligned}$$

In the definition for SUCC, the expression $n\ f\ x$ applies f to x n times (assuming that variable n is the Church encoding of the natural number n). We then apply f to the result, meaning that we apply f to x $n + 1$ times.

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

$$\mathsf{PLUS} \triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \mathsf{SUCC}\ n_2$$