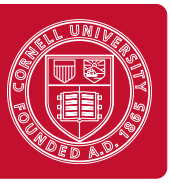
CS 4110 – Programming Languages and Logics
Lecture #15: Programming in λ-calculus



This lecture will build further intuitions for the λ-calculus by exploring encodings using S and K combinators, as well as encodings of common datatypes such as booleans and integers.

1 Combinators

In the last lecture, we saw the DeBruijn representation of λ-calculus expressions. 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:

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

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

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

It is not hard to see that I is not needed—it can be encoded as $\mathsf{S}\ \mathsf{K}\ \mathsf{K}$.

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:

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

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

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

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

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

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.

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

and

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

2 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.

2.1 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.

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

2.2 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.

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