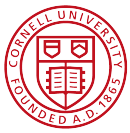
CS 4110 – Programming Languages and Logics
Lecture #16: Encodings



0.1 Evaluation contexts

Recall the syntax and CBV operational semantics for the lambda calculus:

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

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'} \qquad \beta\text{-reduction}\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}$$

Of the operational semantics rules, only the β-reduction rule told us how to "reduce" an expression; the other two rules tell us the order to evaluate expressions—e.g., evaluate the left hand side of an application to a value first, then evaluate the right hand side of an application to a value. The operational semantics of many of the languages we will consider have this feature: there are two kinds of rules, *congruence rules* that specify evaluation order, and the *computation rules* that specify the "interesting" reductions.

Evaluation contexts are a simple mechanism that separates out these two kinds of rules. An evaluation context E (sometimes written $E[\cdot]$) is an expression with a "hole" in it, that is with a single occurrence of the special symbol $[\cdot]$ (called the "hole") in place of a subexpression. Evaluation contexts are defined using a BNF grammar that is similar to the grammar used to define the language. The following grammar defines evaluation contexts for the pure CBV λ-calculus.

$$E ::= [\cdot] \mid E\ e \mid v\ E$$

We write $E[e]$ to mean the evaluation context E where the hole has been replaced with the expression e. The following are examples of evaluation contexts, and evaluation contexts with the hole filled in by an expression.

$$E_1 = [\cdot]\ (\lambda x.\, x) \qquad E_1[\lambda y.\, y\ y] = (\lambda y.\, y\ y)\ \lambda x.\, x$$
$$E_2 = (\lambda z.\, z\ z)\ [\cdot] \qquad E_2[\lambda x.\, \lambda y.\, x] = (\lambda z.\, z\ z)\ (\lambda x.\, \lambda y.\, x)$$
$$E_3 = ([\cdot]\ \lambda x.\, x\ x)\ ((\lambda y.\, y)\ (\lambda y.\, y)) \qquad E_3[\lambda f.\, \lambda g.\, f\ g] = ((\lambda f.\, \lambda g.\, f\ g)\ \lambda x.\, x\ x)\ ((\lambda y.\, y)\ (\lambda y.\, y))$$

Using evaluation contexts, we can define the evaluation semantics for the pure CBV λ-calculus with just two rules, one for evaluation contexts, and one for β-reduction.

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \beta\text{-reduction}\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}$$

Note that the evaluation contexts for the CBV λ-calculus ensure that we evaluate the left hand side of an application to a value, and then evaluate the right hand side of an application to a value before applying β-reduction.

We can specify the operational semantics of CBN λ-calculus using evaluation contexts:

$$E ::= [\cdot] \mid E\ e \qquad \frac{e \to e'}{E[e] \to E[e']} \qquad \beta\text{-reduction}\ \frac{}{(\lambda x.\, e_1)\ e_2 \to e_1\{e_2/x\}}$$

We'll see the benefit of evaluation contexts as we see languages with more syntactic constructs.

0.2 Multi-argument functions and currying

Our syntax for functions only allows function with a single argument: $\lambda x.\, e$. We could define a language that allows functions to have multiple arguments.

$$e ::= x \mid \lambda x_1, \ldots, x_n.\, e \mid e_0\ e_1\ \ldots\ e_n$$

Here, a function $\lambda x_1, \ldots, x_n.\, e$ takes n arguments, with names x_1 through x_n. In a multi argument application $e_0\ e_1\ \ldots\ e_n$, we expect e_0 to evaluate to an n-argument function, and $e_1, \ldots, e_n$ are the arguments that we will give the function.

We can define a CBV operational semantics for the multi-argument λ-calculus as follows.

$$E ::= [\cdot] \mid v_0\ \ldots\ v_{i-1}\ E\ e_{i+1}\ \ldots\ e_n \qquad \frac{e \to e'}{E[e] \to E[e']}$$

$$\beta\text{-reduction}\ \frac{}{(\lambda x_1, \ldots, x_n.\, e_0)\ v_1\ \ldots\ v_n \to e_0\{v_1/x_1\}\{v_2/x_2\}\ldots\{v_n/x_n\}}$$

The evaluation contexts ensure that we evaluate a multi-argument application $e_0\ e_1\ \ldots\ e_n$ by evaluating each expression from left to right down to a value.

Now, the multi-argument λ-calculus isn't any more expressive that the pure λ-calculus. We can show this by showing how any multi-argument λ-calculus program can be translated into an equivalent pure λ-calculus program. We define a translation function $\mathcal{T}[\![\cdot]\!]$ that takes an expression in the multi-argument λ-calculus and returns an equivalent expression in the pure λ-calculus. That is, if e is a multi-argument lambda calculus expression, $\mathcal{T}[\![e]\!]$ is a pure λ-calculus expression.

We define the translation as follows.

$$\begin{aligned}
\mathcal{T}[\![x]\!] &= x \\
\mathcal{T}[\![\lambda x_1, \ldots, x_n.\, e]\!] &= \lambda x_1.\ \ldots\ \lambda x_n.\, \mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_0\ e_1\ e_2\ \ldots\ e_n]\!] &= (\ldots((\mathcal{T}[\![e_0]\!]\ \mathcal{T}[\![e_1]\!])\ \mathcal{T}[\![e_2]\!])\ldots\ \mathcal{T}[\![e_n]\!])
\end{aligned}$$

This process of rewriting a function that takes multiple arguments as a chain of functions that each take a single argument is called *currying*. Consider a mathematical function that takes two arguments, the first from domain A and the second from domain B, and returns a result from domain C. We could describe this function, using mathematical notation for domains of functions, as being an element of $A \times B \to C$. Currying this function produces a function that is an element of $A \to (B \to C)$. That is, the curried version of the function takes an argument from domain A, and returns a function that takes an argument from domain B and produces a result of domain C.

1 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.

1.1 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.

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

1.2 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.

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