Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Lambda calculus encodings; Recursion

Lecture 7 Tuesday, February 18, 2014

1 Lambda calculus 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.

1.1 Booleans

We want to encode constants and operators for booleans. That is, we want to define functions TRUE, FALSE, AND, IF, and other operators such that the expected behavior holds, for example:

$$\begin{aligned}\mathit{AND}\ \mathit{TRUE}\ \mathit{FALSE} &= \mathit{FALSE}\\ \mathit{IF}\ \mathit{TRUE}\ e_1\ e_2 &= e_1\\ \mathit{IF}\ \mathit{FALSE}\ e_1\ e_2 &= e_2\end{aligned}$$

Let's start by defining TRUE and FALSE as follows.

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

Thus, both TRUE and FALSE take take two arguments, TRUE returns the first, and FALSE returns the second.

The function IF should behave like $\lambda b.\,\lambda t.\,\lambda f.$ if $b = \mathit{TRUE}$ then t else f. The definitions for TRUE and FALSE make this very easy.

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

Definitions of other operators are also straightforward.

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

1.2 Church numerals

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

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

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.

$$PLUS \triangleq \lambda n_1.\, \lambda n_2.\, n_1\ SUCC\ n_2$$

2 Nontermination

Consider the expression $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$, which we will refer to as Ω for brevity. Let's try evaluating Ω.

$$\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \longrightarrow (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) = \Omega$$

Evaluating Ω never reaches a value! It is an infinite loop!

What happens if we use Ω as an actual argument to a function? Consider the following program.

$$(\lambda x.(\lambda y.y))\ \Omega$$

If we use CBV semantics to evaluate the program, we must reduce Ω to a value before we can apply the function. But Ω never evaluates to a value, so we can never apply the function. Under CBV semantics, this program does not terminate.

If we use CBN semantics, then we can apply the function immediately, without needing to reduce the actual argument to a value. We have

$$(\lambda x.(\lambda y.y))\ \Omega \longrightarrow_{\text{CBN}} \lambda y.y$$

CBV and CBN are common evaluation orders; many programming languages use CBV semantics. Later we will see Call-by-need semantics, a more efficient semantics similar to CBN in that it does not evaluate actual arguments unless necessary. Other evaluation orders are also possible. For example, we could define semantics that allow β-reduction inside an abstraction.

$$(\lambda x.\, x + 7 + 8)\ 8 \quad\longrightarrow\quad (\lambda x.\, x + 15)\ 8 \quad\longrightarrow\quad 8 + 15 \quad\longrightarrow\quad 23$$

3 Recursion and the fixed-point combinators

We can write nonterminating functions, as we saw with the expression Ω. We can also write recursive functions that terminate. However, one complication is how we express this recursion.

Let's consider how we would like to define a function that computes factorials.

$$FACT \triangleq \lambda n.\, IF\ (ISZERO\ n)\ 1\ (TIMES\ n\ (FACT\ (PRED\ n)))$$

In slightly more readable notation (and we will see next lecture how we can translate more readable notation into appropriate expressions):

$$FACT \triangleq \lambda n.\, \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times FACT\ (n-1)$$

Here, like in the definitions we gave above, the name $FACT$ is simply meant to be shorthand for the expression on the right-hand side of the equation. But $FACT$ appears on the right-hand side of the equation as well! This is not a definition, it's a recursive equation.

3.1 Recursion Removal Trick

We can perform a "trick" to define a function $FACT$ that satisfies the recursive equation above. First, let's define a new function $FACT'$ that looks like $FACT$, but takes an additional argument f. We assume that the function f will be instantiated with an actual parameter of... $FACT'$.

$$FACT' \triangleq \lambda f.\, \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (f\ f\ (n-1))$$

Note that when we call f, we pass it a copy of itself, preserving the assumption that the actual argument for f will be $FACT'$.

Now we can define the factorial function $FACT$ in terms of $FACT'$.

$$FACT \triangleq FACT'\ FACT'$$

Let's try evaluating $FACT$ applied to an integer.

$$\begin{aligned}
FACT\ 3 &= (FACT'\ FACT')\ 3 && \text{Definition of } FACT \\
&= ((\lambda f.\, \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (f\ f\ (n-1)))\ FACT')\ 3 && \text{Definition of } FACT' \\
&\longrightarrow (\lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (FACT'\ FACT'\ (n-1)))\ 3 && \text{Application to } FACT' \\
&\longrightarrow \textsf{\textbf{if}}\ 3 = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ 3 \times (FACT'\ FACT'\ (3-1)) && \text{Application to } n \\
&\longrightarrow 3 \times (FACT'\ FACT'\ (3-1)) && \text{Evaluating } \textsf{\textbf{if}} \\
&\longrightarrow \ldots \\
&\longrightarrow 3 \times 2 \times 1 \times 1 \\
&\longrightarrow^* 6
\end{aligned}$$

So we now have a technique for writing a recursive function f: write a function f' that explicitly takes a copy of itself as an argument, and then define $f \triangleq f'\ f'$.

3.2 Fixed point combinators

There is another way of writing recursive functions: expressing the recursive function as the fixed point of some other, higher-order function, and then finding that fixed point.

Let's consider the factorial function again. The factorial function $FACT$ is a fixed point of the following function.

$$G \triangleq \lambda f.\, \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (f\ (n-1))$$

(Recall that if g if a fixed point of G, then we have $G\ g = g$.)

So if we had some way of finding a fixed point of G, we would have a way of defining the factorial function $FACT$.

There are such "fixed point operators," and the (infamous) Y combinator is one of them. Thus, we can define the factorial function $FACT$ to be simply $Y\ G$, the fixed point of G.

(A *combinator* is simply a closed lambda term; it is a higher-order function that uses only function application and other combinators to define a result from its arguments; our functions $SUCC$ and ADD are examples of combinators. It is possible to define programs using only combinators, thus avoiding the use of variables completely.)

The Y combinator is defined as

$$Y \triangleq \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)).$$

It was discovered by Haskell Curry, and is one of the simplest fixed-point combinators.

We'll use a slight variant of the Y combinator, Z, which is easier to use under CBV. (What happens when we evaluate $Y\ G$ under CBV? Have a try and see.). The Z combinator is defined as

$$Z \triangleq \lambda f.\, (\lambda x.\, f\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, f\ (\lambda y.\, x\ x\ y)).$$

The fixed point of the higher order function G is equal to $G\ (G\ (G\ (G\ (G\ \ldots))))$. Intuitively, the Z combinator unrolls this equality, as needed. Let's see it in action, on our function G, where

$$G = \lambda f.\, \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (f\ (n-1))$$

and the factorial function is the fixed point of G. (We will use CBV semantics.)

$$\begin{aligned}
FACT &= Z\ G \\
&= (\lambda f.\, (\lambda x.\, f\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, f\ (\lambda y.\, x\ x\ y)))\ G && \text{Definition of } Z \\
&\longrightarrow (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y)) \\
&\longrightarrow G\ (\lambda y.\, (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ y) \\
&= (\lambda f.\, \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (f\ (n-1))) \\
&\qquad (\lambda y.\, (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ y) && \text{Definition of } G \\
&\longrightarrow \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1 \\
&\qquad \textsf{\textbf{else}}\ n \times ((\lambda y.\, (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ y)\ (n-1))
\end{aligned}$$

Here, we will reduce $(\lambda y.\, (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ y)\ (n-1)$, producing an expression which is β-equivalent.

$$\begin{aligned}
&=_\beta \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1 \\
&\qquad \textsf{\textbf{else}}\ n \times ((\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (n-1))
\end{aligned}$$

Now note that $(\lambda x.\, G\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, G\ (\lambda y.\, x\ x\ y))$ is β-equivalent to $Z\ G$, allowing us to rewrite the expression as follows.

$$\begin{aligned}
&=_\beta \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times ((Z\ G)\ (n-1)) \\
&= \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (FACT\ (n-1)) && \text{By defn. of factorial}
\end{aligned}$$

There are many (indeed infinite) fixed-point combinators. To gain some more intuition for fixed-point combinators, let's derive the Turing fixed-point combinator, discovered by Alan Turing, and denoted by Θ.

Suppose we have a higher order function f, and want the fixed point of f. We know that $\Theta\ f$ is a fixed point of f, so we have

$$\Theta\ f = f\ (\Theta\ f).$$

This means, that we can write the following recursive equation for Θ.

$$\Theta = \lambda f.\, f\ (\Theta\ f)$$

.

Now we can use the recursion removal trick we described earlier! Let's define $\Theta' = \lambda t.\, \lambda f.\, f\ (t\ t\ f)$, and define

$$\begin{aligned}
\Theta &\triangleq \Theta'\ \Theta' \\
&= (\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f))
\end{aligned}$$

Let's try out the Turing combinator on our higher order function G that we used to define $FACT$. This time we will use CBN semantics.

$$\begin{aligned}
FACT &= \Theta\ G \\
&= ((\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f)))\ G \\
&\longrightarrow (\lambda f.\, f\ ((\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ f))\ G \\
&\longrightarrow G\ ((\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ (\lambda t.\, \lambda f.\, f\ (t\ t\ f))\ G) \\
&= G\ (\Theta\ G) && \text{for brevity} \\
&= (\lambda f.\, \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (f\ (n-1)))\ (\Theta\ G) && \text{Definition of } G \\
&\longrightarrow \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times ((\Theta\ G)\ (n-1)) \\
&= \lambda n.\, \textsf{\textbf{if}}\ n = 0\ \textsf{\textbf{then}}\ 1\ \textsf{\textbf{else}}\ n \times (FACT\ (n-1))
\end{aligned}$$