CS 321 Programming Languages

Intro to Lambda Calculus

Baris Aktemur

Özyeğin University

Last update made on Monday 11th December, 2017 at 14:36.

Lambda Calculus

In 1930's, mathematicians were looking for a foundational calculus that would allow them study computability. They came up with Lambda Calculus, whose syntax is given below.

$$
\begin{aligned}
x &\in Var \\
e &\in Exp ::= x \mid e\ e \mid \lambda x.e
\end{aligned}
$$

Lambda calculus is able to express *anything* that's computable. This means, anything you write in Java, C, Python, etc. can be expressed in lambda calculus. I find this fact mind-blowing.

Lambda calculus is equivalent to the universal Turing machine; either can be used to model computable functions.

Pioneers of lambda calculus include Alonzo Church and Haskell Curry. Spend some time to read about them.

Lambda calculus

There are three constructs in Lambda calculus:

1. Variables (e.g. x, y, z). They come from an infinite set.
2. Function application, $e_1\ e_2$. You're already familiar with this.
3. Lambda abstraction, $\lambda x.e$. This is the same as anonymous functions in OCaml, e.g. `fun` x `->` e.

β-reduction

In lambda calculus, terms are reduced using β-reduction, as defined below.

$$(\lambda x.e_1)e_2 \Rightarrow [x/e_2]e_1$$

where $[x/e_2]e_1$ means "substitute every free occurrence of x in e_1 with e_2. For instance

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

Or, a slightly bigger example where the reduced term is underlined:

$$
\begin{aligned}
& \underline{(\lambda f.\lambda x.fx)(\lambda y.y)}(\lambda z.zz) \\
\Rightarrow\ & (\lambda x.\underline{(\lambda y.y)x})(\lambda z.zz) \\
\Rightarrow\ & \underline{(\lambda x.x)(\lambda z.zz)} \\
\Rightarrow\ & \lambda z.zz
\end{aligned}
$$

Normal form

When there does not exist any opportunities for β-reduction, a term is said to be in *normal form*. The previous example showed a way to reach the normal form $\lambda z.zz$ from the original term $(\lambda f.\lambda x.fx)(\lambda y.y)(\lambda z.zz)$. In fact, there exist another order of reductions to reach the same normal form:

$$\begin{aligned} & \underline{(\lambda f.\lambda x.fx)(\lambda y.y)}(\lambda z.zz) \\ \Rightarrow\ & \underline{(\lambda x.(\lambda y.y)x)(\lambda z.zz)} \\ \Rightarrow\ & \underline{(\lambda y.y)(\lambda z.zz)} \\ \Rightarrow\ & \lambda z.zz \end{aligned}$$

Confluence

A very strong and important theorem (due to Church and Rosser) states that for a term, there exists at most one normal form. This means, if there is a normal form of a term, no matter the order of reductions, you will eventually reach that normal form.

Note that there may not exist a normal form of a term. A well-known example is the famous ω (omega) term, which inifinitely reduces to itself:

$$\begin{aligned} & (\lambda x.xx)(\lambda x.xx) \\ \Rightarrow\ & (\lambda x.xx)(\lambda x.xx) \\ \Rightarrow\ & (\lambda x.xx)(\lambda x.xx) \\ \Rightarrow\ & \ldots \end{aligned}$$

Church numerals

At the beginning of this lecture, we stated that lambda calculus can express anything that's computable. The lambda calculus syntax does not include integers, addition, multiplication, if-expressions, etc. All of these are encodable in lambda calculus. The following is an encoding of natural numbers in lambda calculus, known as the Church numerals:

$$\begin{aligned} \mathbf{0} &= (\lambda f.\lambda x.x) \\ \mathbf{1} &= (\lambda f.\lambda x.fx) \\ \mathbf{2} &= (\lambda f.\lambda x.f(fx)) \\ \mathbf{3} &= (\lambda f.\lambda x.f(f(fx))) \\ \mathbf{4} &= (\lambda f.\lambda x.f(f(f(fx)))) \end{aligned}$$

and so on.

Church numerals

Then, the successor function, which takes a Church numeral and returns the next Church numeral, is defined as follows:

$$\textbf{succ} = \lambda n.\lambda f.\lambda x.f(nfx)$$

Similarly, addition and multiplication functions, which take two Church numerals and return, respectively, their sum and product, are defined below:

$$\begin{aligned} \textbf{add} &= \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx) \\ \textbf{mult} &= \lambda m.\lambda n.\lambda f.\lambda x.m(nf)x \end{aligned}$$

Church numerals

As an example, let's show that **succ 1** $=$ **2**.

$$
\begin{aligned}
& \textbf{succ 1} \\
&= (\lambda n.\lambda f.\lambda x.f(nfx))\mathbf{1} \\
&\Rightarrow \lambda f.\lambda x.f(\mathbf{1}fx) \\
&= \lambda f.\lambda x.f(\underline{(\lambda f.\lambda x.fx)f}x) \\
&\Rightarrow \lambda f.\lambda x.f(\underline{(\lambda x.fx)x}) \\
&\Rightarrow \lambda f.\lambda x.f(fx) \\
&= \mathbf{2}
\end{aligned}
$$

Church numerals

Let's also show that **add 1 2** $=$ **3**.

$$
\begin{aligned}
& \textbf{add 1 2} \\
&= (\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx))\mathbf{1}\ \mathbf{2} \\
&\Rightarrow (\lambda n.\lambda f.\lambda x.\mathbf{1}f(nfx))\mathbf{2} \\
&\Rightarrow \lambda f.\lambda x.\mathbf{1}f(\mathbf{2}fx) \\
&= \lambda f.\lambda x.\underline{(\lambda f.\lambda x.fx)f}(\mathbf{2}fx) \\
&\Rightarrow \lambda f.\lambda x.\underline{(\lambda x.fx)(\mathbf{2}fx)} \\
&\Rightarrow \lambda f.\lambda x.(f(\mathbf{2}fx)) \\
&= \lambda f.\lambda x.f(\underline{(\lambda f.\lambda x.f(fx))f}x) \\
&\Rightarrow \lambda f.\lambda x.f(\underline{(\lambda x.f(fx))x}) \\
&\Rightarrow \lambda f.\lambda x.f(f(fx)) \\
&= \mathbf{3}
\end{aligned}
$$

Church numerals

Here is the encoding for the **pred** function that is the dual of **succ**; it returns the predecessor of the given number.

$$\textbf{pred} = \lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda u.u)$$

For instance, **pred (add 2 3)** now gives you the lambda term corresponding to **4**.

Note: The definition of **pred** is quite difficult to comprehend. You do not need to spend too much time understanding how it could be derived.

Booleans

Here is the encoding for booleans and two useful functions.

$$
\begin{aligned}
\textbf{true} &= \lambda a.\lambda b.a \\
\textbf{false} &= \lambda a.\lambda b.b \\
\textbf{if} &= \lambda c.\lambda t.\lambda e.\ c\,t\,e \\
\textbf{isZero} &= \lambda n.n(\lambda x.\textbf{false})\textbf{true}
\end{aligned}
$$

Implementation

Now see these encodings in action at

```
https://github.com/aktemur/cs321/tree/master/Lambda
```


Recursion

An important question you may ask is how to encode recursion (since we don't have let/let-rec bindings in lambda calculus). Let's begin by an attempt to define the factorial function.

$$\textbf{fact} = \lambda \text{m}.\textbf{if}(\textbf{isZero}\ \text{m})(\mathbf{1})(\textbf{mult}\ \text{m}\ (\textbf{fact}(\textbf{pred}\ \text{m})))$$

In this definition, there is circularity; **fact** depends on its own definition. We may attempt to make the definition a closed, pure lambda calculus term, by substituting **fact** with its definition, but this does not work because it leads to infinite expansion. So what to do?

Fixed points

Let's have a short pause and give a definition:

Definition

Given a function f and a value x, it is said that x is a fixed point of f if $f(x) = x$.

For example, 3 is a fixed point of $f(x) = x^2 - 6$ because $f(3) = 3$.

Recursion

Let's go back to our definition of the factorial function. To fix the circular definition problem, let's make the factorial function receive the recursive function as a parameter.

$$\mathbf{F} = \lambda \text{fact}.\lambda \text{m}.\textbf{if}(\textbf{isZero}\ \text{m})(\mathbf{1})(\textbf{mult}\ \text{m}\ (\text{fact}(\textbf{pred}\ \text{m})))$$

Now, **F** is a closed, valid lambda expression. If we were able to apply **F** on the **fact** function, we would get the factorial function. That is:

$$\mathbf{F(fact)} = \textbf{fact}$$

Hey, this means **fact** is a fixed point of **F**. If we can find the fixed point of **F**, we can find a proper definition for **fact**.

Recursion

Suppose we have a function **fix** that finds the fixed point of a given function. We could then define **fact** as

$$\textbf{fact} = \textbf{fix F}$$

Fortunately, there exist infinitely many fixed point calculators (called fixed point combinators) in lambda calculus. The most famous is the Y-combinator[1] (due to Haskell Curry):

$$\mathbf{Y} = \lambda g.(\lambda x.g(xx))(\lambda x.g(xx))$$

As an exercise, compute **fact 2**. Also read the Wikipedia article: `http://en.wikipedia.org/wiki/Fixed-point_combinator`.

[1]There also is a company with this name that provides seed funding to startups. See `http://ycombinator.com`.

Note

A final note: the encodings we've seen here work in untyped lambda calculus. There also exist typed versions of lambda calculus. In a simply typed setting, recursion and many terms such as ω can't be written because they don't type-check. Also, **Y**-combinator does not work under call-by-value semantics because it diverges (i.e. causes infinite reductions). When using call-by-value semantics, another fixed point combinator must be used. See PLC Section 5.6 for an example.

Lambda calculus

One language to rule them all