Lecture 14: λ Calculus II

built on 2021/02/18 at 09:56:05

We will develop the familiar pieces in lambda calculus to enable programming in it.

1 The Church Numerals

We now show how to represent natural numbers using no additional constructs. One way to encode a number n is to write a λ expression that repeats something n times. Indeed, the Church numeral—due to Alonzo Church, the inventor of lambda calculus—is based on this idea, where the n-th Church numeral, written $\overline{n}$, is defined as

$$\overline{n} = \lambda f x.\, f^n\, x.$$

Hence, the first few Church numerals look as follows:

$$\overline{0} = \lambda f x.\, x \qquad \overline{3} = \lambda f x.\, f(f(f\, x))$$
$$\overline{1} = \lambda f x.\, f\, x \qquad \overline{4} = \lambda f x.\, f(f(f(f\, x)))$$
$$\overline{2} = \lambda f x.\, f(f\, x) \qquad \overline{5} = \lambda f x.\, f(f(f(f(f\, x))))$$

This encoding is reminiscent of the unary encoding, in which we use n bits to represent the number n.

Having defined the encoding, we can start implementing functions that operate on it. First, how do we increment a number? The successor function **succ** takes a lambda expression representing a number n and evaluates to a lambda expression representing $n+1$—that is to say, $\mathbf{succ}\,\overline{n} \to_* \overline{n+1}$. This can be implemented as follows:

$$\mathbf{succ} = \lambda n f x.\, f(n f x).$$

Intuitively, the successor function wraps $\overline{n}$ with one more f at the outermost location. It is a simple exercise to verify that for any $n \geqslant 0$, $\mathbf{succ}\,\overline{n} \to_* \overline{n+1}$.

The predecessor function **pred**, which decrements the number by 1, exists but is surprisingly more difficult to derive. See the exercises at the end of this chapter for a step-by-step guide.

Before we move on, let's look at how we can support addition and multiplication. Back in grade school, we learned that $m+n$ is simply incrementing m a total of n times. For example, $m+3$ is $m+1+1+1$. This motivates the following implementation of **add**:

$$\mathbf{add} = \lambda m n f x.\, n\, f\, (m\, f\, x),$$

which essentially says wraps m with f a total n times. By a similar reasoning, we can derive the multiplication function as:

$$\mathbf{mult} = \lambda m n f.\, n(m\, f)$$

To see why these implementations work, it helps to try to manually step through the execution of, for example, $\mathbf{add}\,\overline{2}\,\overline{3}$ and $\mathbf{mult}\,\overline{2}\,\overline{3}$.

More formally, the specifications are: for $m, n \geqslant 0$ are natural numbers,

$$\mathbf{add}\,\overline{n}\,\overline{m} \to_* \overline{n+m}$$
$$\mathbf{mult}\,\overline{n}\,\overline{m} \to_* \overline{n \times m}$$

It is possible to implement **iszero** such that

$$\mathbf{iszero}(n) = \begin{cases} \mathbf{T} & \text{if } n = 0 \\ \mathbf{F} & \text{otherwise, i.e., } n > 0. \end{cases}$$

(We leave this as an exercise to the reader.)

Armed with these, we can compose these functions together to express computation such as $f(x) = 2x + 5$ as $\mathbf{f} = \lambda x.\, \mathbf{add}\,(\mathbf{mult}\,\overline{2}\,x)\,\overline{5}$.

2 Fixed-Point Theorem and Recursion

Experienced programmers can use recursion to naturally express code that is otherwise clumsy to write. In this section, we will show how to obtain recursion from the mechanisms we have built so far.

The factorial function, given by $n! = n \times (n-1)!$ with $0! = 1$, is a prototypical example of a recursive function. If we implement the factorial function recursively in Python, we would end up with more or less the following code:

```
def fac(n: int): -> int
    if n == 0:
        return 1
    return fac(n-1) * n
```

Notice that to write a recursive function, we need to be able to reference the very function that we are writing. In the code above, we refer to `fac` in the body of `fac`. This turns out to be an important challenge in our setup of lambda calculus because in this setup, the act of defining a function is separated from naming that function, resulting in a situation where while we are defining the function, we cannot refer to this very function by any name. We will fix this using the concept of a fixed point.

Fixed-Point. In Math, a value x is a fixed point of a function f if $f(x) = x$. That is to say, the function returns the value that was passed as its argument. For example, $x = 0$ and $x = 1$ are the only fixed points of $f(x) = x^2$. But for $g(x) = x$, every $x \in \mathbb{R}$ is a fixed point. Yet, functions such as $h(x) = x + 1$ have no fixed point. As these examples show, not every function has a fixed point in arithmetic and Calculus.

In lambda calculus, if F and N are lambda terms, we say that N is a fixed point of F if $N \to_* F\,N$. In sharp contrast with the arithmetic setting, every lambda term F has a fixed point—and this is what we are going to leverage to derive recursion.

Theorem 2.1 (Fixed-Point Theorem) *Every lambda term F has a fixed point. Specifically, there exists a lambda term Θ such that for every term F, the term $N = \Theta\,F$ is a fixed point of F.*

Proof: Define $A = \lambda xy.\, y\,(xxy)$ and $\Theta = AA$. Then:

$$\begin{aligned} N &\to \Theta\,F \\ &\to AA\,F \\ &\to (\lambda xy.\, y\,(xxy))\,AF \\ &\to F\,(AA\,F) \\ &\to F\,(\Theta\,F) \\ &\to F\,N, \end{aligned}$$

where we note that the Θ term here is known in the literature as Turing's fixed point combinator[1]. ∎

Deriving Recursion. To see how we can apply the fixed-point theorem (Theorem 2.1) to derive a recursive function, consider once again the factorial example from before. Suppose Python lacked the ability to self-reference a function. The previous code could be rewritten as follows:

```
def plain_fac(F: Function[int, int], n: int): -> int
    if n == 0:
        return 1
    return F(n - 1)*n
```

[1] There are a couple other combinators such as Y-combinator.

This code has a curious parameter `F`, which is a function from an `int` to an `int`. We would be all set if there is a function `F` satisfying `F(n) = plain_fac(F, n)` for all `n`. Does this smell like a fixed point on functions? It indeed is.

To see this more clearly, let us now write `plain_fac` in lambda calculus:

$$\textbf{plain\_fac} = \lambda f.\, \lambda n.\, \textbf{if\_then\_else}\,(\textbf{iszero}\,n)\,(\overline{1})\,(\textbf{mult}\,(f\,(\textbf{pred}\,n))\,n)$$

At this point, we can easily check that if we have a lambda term F that is a fixed point of **plain_fac**, then that lambda term F is a recursive implementation of factorial. In other words, for any n,

$$\begin{aligned} F\,n &\to_* (\textbf{plain\_fac}\,F)\,n \\ &\to_* \textbf{if\_then\_else}\,(\textbf{iszero}\,n)\,(\overline{1})\,(\textbf{mult}\,(F\,(\textbf{pred}\,n))\,n). \end{aligned}$$

How can we obtain such a fixed point? We can readily apply the fixed-point theorem. If we let $F = \Theta\,\textbf{plain\_fac}$, then we can apply F to $\overline{2}$ and obtain the following:

$$\begin{aligned}
F\,\overline{2} &\to_* (\textbf{plain_fac}\,F)\,\overline{2} && \text{[fixed-point thm]} \\
&\to_* \textbf{if_then_else}\,(\textbf{iszero}\,\overline{2})\,(\overline{1})\,(\textbf{mult}\,(F\,(\textbf{pred}\,\overline{2}))\,\overline{2}) \\
&\to_* \textbf{if_then_else}\,(\textbf{F})\,(\overline{1})\,(\textbf{mult}\,(F\,(\textbf{pred}\,\overline{2}))\,\overline{2}) \\
&\to_* \textbf{mult}\,(F\,(\textbf{pred}\,\overline{2}))\,\overline{2} \\
&\to_* \textbf{mult}\,(F\,\overline{1})\,\overline{2} \\
&\to_* \textbf{mult}\,((\textbf{plain_fac}\,F)\,\overline{1})\,\overline{2} && \text{[fixed-point thm]} \\
&\to_* \textbf{mult}\,(\textbf{if_then_else}\,(\textbf{iszero}\,\overline{1})\,(\overline{1})\,(\textbf{mult}\,(F\,(\textbf{pred}\,\overline{1}))\,\overline{1}))\,\overline{2} \\
&\vdots \\
&\to_* \textbf{mult}\,(\textbf{mult}\,(F\,\overline{0})\,\overline{1})\,\overline{2} \\
&\to_* \textbf{mult}\,(\textbf{mult}\,((\textbf{plain_fac}\,F)\,\overline{0})\,\overline{1})\,\overline{2} && \text{[fixed-point thm]} \\
&\to_* \textbf{mult}\,(\textbf{mult}\,(\textbf{if_then_else}\,(\textbf{iszero}\,\overline{0})\,(\overline{1})\,(\textbf{mult}\,(F\,(\textbf{pred}\,\overline{0}))\,\overline{0}))\,\overline{1})\,\overline{2} \\
&\to_* \textbf{mult}\,(\textbf{mult}\,\overline{1}\,\overline{1})\,\overline{2} \\
&\to_* \textbf{mult}\,(\overline{1})\,\overline{2} \\
&\to_* \overline{2},
\end{aligned}$$

correctly producing $\overline{2}$, which the result of computing 2!. Notice that this derivation, although tedious, is straightforward. Readers should pay close attention to the steps where fixed-point theorem was applied and what happened in those steps.

3 Exercises

1. Give a lambda expression for **is_zero** whose specification is

$$\begin{aligned} &\textbf{is\_zero}\,\overline{0} \to_* \textbf{T} \\ &\textbf{is\_zero}\,\overline{n} \to_* \textbf{F} \text{ for } n > 0 \end{aligned}$$

2. Consider the following recursive function in Python:

```
def afac(n: int, a: int) -> int:
    if n == 0:
        return a
    else:
        return afac(n-1, a*n)
```

Derive a lambda expression representing a recursive implementation of `afac`. Show what happens when we run it in `afac(2, 1)`.