Lecture 7: Church Numerals and Closures

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 Church Numerals

- Now our goal is to come up with a λ-calculus encoding for numbers, building on our intuition for compiling Booleans from last time
- Let's think about how we can encode the natural numbers $0, 1, 2, \cdots$
- A natural way is to use *nested function calls*: zero nestings is 0, 1 nesting is 1, etc.
- Concretely, we can define the following **Church Numerals** that encode these numbers as particular λ-terms:

$$\begin{aligned}
\texttt{zero} &= \lambda f.\lambda x.x \\
\texttt{one} &= \lambda f.\lambda x.(f\ x) \\
\texttt{two} &= \lambda f.\lambda x.(f\ (f\ x)) \\
\texttt{three} &= \lambda f.\lambda x.(f(f\ (f\ x))) \\
&\cdots
\end{aligned}$$

- This seems natural: `zero` calls f zero times, `one` calls f one time, etc.
- Now, how do we encode various operations on numbers like addition, multiplication, etc.?
- Let's start with a (maybe not-so) easy case: adding 1.
- We want to define a function `succ` (short for "sucessor") that takes in a Church-encoded numeral and returns the next one in the sequence. For example, we want:

```
(succ zero) = one
 (succ one) = two
                ...
```

2 Motivating Equivalence of λ-calculus Programs

- Program equivalence is an essential idea in programming languages, and it's used all the time. It's how we discuss compiler optimizations, compare algorithms, etc.
- The most basic level of equality is **syntactic equality**: two lambda terms are syntactically equivalent if their syntax is equal as strings.
- Syntactic equality is certainly a reasonable notion of equality, but it seems too restrictive. For instance, the following two terms are not syntactically equivalent:

  $$e_1 = \lambda x.x \qquad e_2 = \lambda y.y \tag{1}$$

  Both of these programs clearly represent the same function: they represent the identity function. So, we would like a more general *semantic notion of equality* that can capture this subtlety.
- Clearly we can go on forever, coming up with more and more complicated equivalent λ-terms that differ slightly from each other with different applications of identity functions etc. What we need is a very rich notion of equality that elides all details about how these terms are syntactically represented.
- We want a *semantic notion of equality*. We say two lambda terms are **extensionally equivalent** (also called **observationally equivalent** and **behaviorally equivalent**) if there is no way to tell them apart by calling them. We will make this precise, but first, let's see some examples to understand what kind of terms we want to be observationally equivalent and what kinds we don't.
- Notation: If two terms e_1 and e_2 are observationally equivalent to each other we write this as $e_1 \cong e_2$.
- For instance, the following two terms should be extensionally equivalent, since they both behave exactly like the identity function:

  $$((\lambda x.x)\ (\lambda x.x)) \cong \lambda x.x$$

- These two programs should not be extensionally equivalent, since one loops forever and the other immediately terminates:[1]

  $$\Omega = (\lambda x.x\ x)(\lambda x.x\ x) \ncong \lambda x.x$$

- The following two terms should not be extensionally equivalent, since intuitively they are functions taking different arguments, which we intuitively expect to not be semantically equivalent:

  $$(\lambda x.x) \ncong \lambda x.\lambda y.x \tag{2}$$

- Here is an interesting case: should the following term be extensionally equivalent to `two`?

  $$\texttt{maybe-two} : \lambda f.\lambda x.(f\ (\underbrace{(\lambda f.\lambda x.f\ x)}_{\texttt{one}}\ f\ x))$$

  Let's examine it carefully and see. Our intuition is that any function that behaves just like `two` should take two arguments f and x and call f twice on x. Well, we can see that this `maybe-two` function does exactly that: first, it calls `one` with the arguments f and x; this simply evaluates to $f\ x$. Then, f is called on this (notice the extra f before the underlined `one`). So, we conclude that we want it to be the case that `maybe-two` $\cong$ `two`. Walk through this slowly!

[1]Note I am dropping some parenthesis here for clarity.

3 Defining Equality

- We need to make precise what it means to "tell them apart two functions by calling them".
- Let's start with a basic assertion: a program that runs forever is not observationally equivalent to one that terminates. We say that non-termination and termination are **observationally inequivalent**.
- This seems uncontrovertible. Let's try to use this as our way of telling two functions apart:

  > Two terms are e_1 and e_2 are **observationally equivalent** if there is no way to call both of them with the same argument that results in different terminating behavior.

- At first this definition may seem surprising! How can terminating behavior be a strong enough notion to capture equality?
- Let's see if it can handle the examples we discussed earlier.
- Clearly, $((\lambda x.x)(\lambda x.x)) \cong (\lambda x.x)$ under this definition, since they both immediately evaluate to the identity function whenever they are called.
- What about the interesting case Equation 2? How can we call these two functions with with the same arguments to result in different terminating behavior?
- Let's define a special function that loops forever if you call it:

  $$\texttt{poison-pill} = \lambda w.\Omega = \lambda w.(\lambda x.(x\ x))\ (\lambda x.(x\ x))$$

  Let's use this poison pill function to make these two terms have different observable behavior (here we use `id` as shorthand for $\lambda x.x$):

  $$(((\lambda x.x)\ \texttt{poison-pill})\ \texttt{id})$$

  This runs forever (check for yourself! see why, plug it into the λ-calculus interpreter). How about the other one?

  $$(((\lambda x.\lambda y.x)\ \texttt{poison-pill})\ \texttt{id})$$

  This doesn't run forever! Check: what does it evaluate to? Why doesn't this program run forever?

- So, we can conclude from the above that $(\lambda x.x) \ncong (\lambda x.\lambda y.x)$, since we found identical arguments to call each with that result in different terminating behavior.
- Note: In many ways, it is much easier to show two terms are inequivalent than it is to show they are equivalent! To show two terms are inequivalent, we "just" have to find some arguments that make them terminate differently; showing they are equivalent requires arguing about *all possible arguments*. We will not formally show two terms equivalent in this class since it typically requires formal proof techniques like mathematical induction; we will rely on intuitive arguments about equivalence here.

4 Successor Functions

- Now we are ready to write the specification for our successor function: the successor function `succ` is a function that takes a Church numeral as an argument and returns a λ-term that is extensionally equivalent to the next Church numeral.
- So, `succ` must be a lambda term that takes in one argument, which is a Church numeral, and it must return another Church numeral (i.e., function taking two arguments). We can start to fill in this template:

$$\texttt{succ} = \lambda n.\lambda f.\lambda x.???$$

- We need to think carefully to make progress here. What do we want to do to the Church numerals so that `succ` satisfies its specification?
- Intuitively, we need to "add another call to f to the innermost body". We need to do this by *calling the Church numeral n in a specific way*, somewhat similar to how we implemented `ite` by calling the Church-encoded Boolean in a certain way
- Let's try some things. Here is a candidate:

$$\texttt{maybe-succ} = \lambda n.\lambda f.\lambda x.(n\ f)$$

What does `maybe-succ` look like? Let's see what happens when we invoke it on `zero`:

$$(\texttt{maybe-succ one}) = \Big((\lambda n.\lambda f.\lambda x.(n\ f))\ (\lambda f.\lambda x.(f\ x))\Big)$$

This term runs to $(\lambda f.\lambda x.((\lambda f.\lambda x.(f\ x))\ f))$. Is this right? Look at it very carefully: there is only a single call to f in this! That doesn't seem right.

- Let's try a different option:

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

- Let's try running it on `one`:

$$(\texttt{succ one}) = \Big((\lambda n.\lambda f.\lambda x.\ (f\ ((n\ f)\ x)))\ (\lambda f.\lambda x.f\ x)\Big) \Downarrow \lambda f.\lambda x.(f\ (\underbrace{(\lambda f.\lambda x.f\ x)}_{\texttt{one}}\ f\ x))$$

(Walk through this derivation yourself, or use the λ-calculus interpreter to see this). This is our `maybe-two` example from earlier! This is why we had to introduce all that machinery about equivalence: it is often the case that Church-encoded operations only produce observationally-equivalent versions of the encoded values.