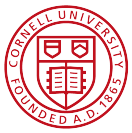
CS 4110 – Programming Languages and Logics
Lecture #16: Fixed-Point Combinators



1 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:

$$\begin{array}{rcl} \Omega & = & (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\ & \to & (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\ & = & \Omega \end{array}$$

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 this 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. Thus, under CBV semantics, this program does not terminate. If we use CBN semantics, we can apply the function immediately, without needing to reduce the actual argument to a value:

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

This difference highlights the fact that evaluation orders, like CBV and CBN, can differ in terms of termination: expressions that diverge under one evaluation order may terminate in another.

2 Recursion

Ω demonstrates that we can write nonterminating λ-terms, but so far they are not very useful. Can we write *recursive* functions in the λ-calculus that do actually terminate and produce a value, as we can in most "real" programming languages? It seems difficult at first because all functions are anonymous—there is no name that a function can use to refer to itself. Surprisingly, however, it is possible to make functions that behave recursively. Roughly, our strategy will build on the "self-replicating" behavior to let functions invoke copies of themselves an unbounded number of times.

Let's consider how we would like to write a factorial function:

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

In slightly more readable notation, this is just:

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

Here, as in the definition 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! Trying to expand the definition would yield an infinite term. This is not a definition, it's a recursive equation.

2.1 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 FACT′ itself.

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

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′.

$$\mathsf{FACT} \triangleq \mathsf{FACT}'\ \mathsf{FACT}'$$

Let's try evaluating FACT on an integer.

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

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 $f \triangleq f'\ f'$.

2.2 Fixed point combinators

There is another way of writing recursive functions: we can express the recursive function as the fixed point of some other, higher-order function, and then take its fixed point. We saw this technique earlier in the course when we defined the denotational semantics for **while** loops.

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

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

Recall that if g is 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 a number of "fixed point combinators," and the (infamous) Y combinator is one of them. Thus, we can define the factorial function FACT to be simply $\mathsf{Y}\ G$, the fixed point of G. The Y combinator is defined as

$$\mathsf{Y} \triangleq \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)).$$

It was discovered by Haskell Curry, and is one of the simplest fixed-point combinators. Note how similar its defnition is to Ω.

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

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

Let's see it in action, on our function G. Define FACT to be $\mathsf{Z}\ G$ and calculate as follows:

$$
\begin{array}{lll}
 & \mathsf{FACT} & \\
= & \mathsf{Z}\ G & \\
= & (\lambda f.\,(\lambda x.\,f\ (\lambda y.\,x\ x\ y))\ (\lambda x.\,f\ (\lambda y.\,x\ x\ y)))\ G & \text{Definition of Z} \\
\longrightarrow & (\lambda x.\,G\ (\lambda y.\,x\ x\ y))\ (\lambda x.\,G\ (\lambda y.\,x\ x\ y)) & \\
\longrightarrow & G\ (\lambda y.\,(\lambda x.\,G\ (\lambda y.\,x\ x\ y))\ (\lambda x.\,G\ (\lambda y.\,x\ x\ y))\ y) & \\
= & (\lambda f.\,\lambda n.\,\textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (f\ (n-1))) & \\
 & \quad (\lambda y.\,(\lambda x.\,G\ (\lambda y.\,x\ x\ y))\ (\lambda x.\,G\ (\lambda y.\,x\ x\ y))\ y) & \text{definition of } G \\
\longrightarrow & \lambda n.\,\textbf{if}\ n = 0\ \textbf{then}\ 1 & \\
 & \quad \textbf{else}\ n \times ((\lambda y.\,(\lambda x.\,G\ (\lambda y.\,x\ x\ y))\ (\lambda x.\,G\ (\lambda y.\,x\ x\ y))\ y)\ (n-1)) & \\
=_\beta & \lambda n.\,\textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (\lambda y.\,(\mathsf{Z}\ G)\ y)\ (n-1) & \\
=_\beta & \lambda n.\,\textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (\mathsf{Z}\ G\ (n-1)) & \\
= & \lambda n.\,\textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times (\mathsf{FACT}\ (n-1)) &
\end{array}
$$

There are many (indeed infinitely many) fixed-point combinators. Here's a cute one:

$$\mathsf{Y}_k \triangleq (\mathsf{L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L\ L})$$

where

$$\mathsf{L} \triangleq \lambda abcdefghijklmnopqstuvwxyzr.\,(r\ (t\ h\ i\ s\ i\ s\ a\ f\ i\ x\ e\ d\ p\ o\ i\ n\ t\ c\ o\ m\ b\ i\ n\ a\ t\ o\ r))$$

To gain some more intuition for fixed-point combinators, let's derive a fixed-point combinator that was originally discovered by Alan Turing. 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:

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

Now we can use the recursion removal trick we described earlier. Define Θ' as $\lambda t.\,\lambda f.\,f\ (t\ t\ f)$, and Θ as $\Theta'\ \Theta'$. Then we have the following equalities:

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

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 evaluation.

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