The Y-Combinator and the Recursion Theorem

Sam Hopkins

November 30, 2012

It seems to be frequently mentioned on the Internet that the Y combinator and the recursion theorem are informally very similar, but I have not been able to find a source for further exposition of the relationship. We will provide such an exposition. *Prerequisites:* basic understanding of recursive functions and lambda calculus.

We will denote recursive functions[1] φ_i. For the remainder of this note, fix a computable enumeration of the recursive functions $\varphi_1, \varphi_2, \ldots$. We denote equivalence functions by $\varphi_i \cong \varphi_j$, meaning that φ_i and φ_j have the same behavior on all inputs. The plan of attack is as follows. We will state the recursion theorem, examine how the fixed points it provides can be used to make a function "call itself", and then examine its proof to see how we might find those fixed points. The resulting fixed-point-finder will be the Y-combinator.

Theorem (Recursion Theorem)**.** *Let f be total recursive. Then there is $n \in \mathbb{N}$ so that $\varphi_n \cong \varphi_{f(n)}$.*

How is this helpful in making functions "call themselves"? The easiest explanation is via an example. Suppose I want to write the factorial function:

$$\texttt{fact(n)} = \texttt{if n} = 0 \texttt{ then } 1 \texttt{ else n} * \texttt{fact(n} - 1\texttt{)}$$

but I don't have named functions. I can write instead the function F:

$$\lambda f.\, \lambda n.\ \texttt{if } n = 0 \texttt{ then } 1 \texttt{ else } n * f(n-1)$$

The recursion theorem claims the existence of some natural number in the presence of a total recursive function. Our function F is total recursive, but natural numbers don't seem to be of much use. The key observation is that when working with actual programs we get to sweep under the rug the distinction between φ_n and its Gödel number n: both are just functions. So in this context, we get from the recursion theorem a function f so that $F(f) \cong f$. On inspection, such an f must be the factorial function. Now we just need to find it.

Proof of the Recursion Theorem. We follow Soare's *Recursively Enumerable Sets and Degrees*. The proof is a diagonalization with a trick: we delay the computation of the diagonal function to make sure that we maintain totality where we need it.

Define the diagonal function $d(u)$ so that

$$\varphi_{d(u)}(n) = \begin{cases} \varphi_{\varphi_u(u)}(n) & \text{if } \varphi_u(u) \text{ converges} \\ \text{diverges} & \text{otherwise} \end{cases}$$

Note that d is total (although the functions it outputs are not; indeed they never converge when $\varphi_u(u)$ doesn't converge). Let f be total recusive. Then since $f \circ d$ is recursive, there is $v \in \mathbb{N}$ so

[1] in Gödel's sense, not in the sense that they "call themselves," although of course that is what the Y-Combinator will enable

that $\varphi_v \cong f \circ d$. We claim that $d(v)$ is a fixed-point for f. The only observation required is that since f and d are total, so is $f \circ d$, so $\varphi_v(v)$ converges. We have

$$\varphi_{d(v)} \cong \varphi_{\varphi_v(v)} \cong \varphi_{\varphi_{(f\circ d)(v)}} \cong \varphi_{f(d(v)}.$$

□

The proof of the recursion theorem reveals how to find fixed-points: we just need to compute $d(f \circ d)$. The only difficulty in translating this to a λ-term is that we must be careful to maintain a distinction between application of a λ-term to another λ-term and precomposing one with the other.

Notice that the diagonal function above corresponds to the λ-term

$$\lambda g.\, \lambda x.\, g\, g\, x.$$

Given an f, we want to *precompose* this with it. The precomposition is given by

$$\lambda y.\, f\, ((\lambda g.\, \lambda x.\, g\, g\, x)\, y)$$

or, unrolling one application (β-reducing),

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

We are ready to write the Y-combinator. Given f, we want to apply the diagonal function to the above term. We get

$$\lambda f.\, (\lambda g.\, \lambda x.\, g\, g\, x)\, (\lambda y.\, f\, (\lambda x.\, y\, y\, x))$$

Recalling that we can β-reduce with a call-by-name semantics if we so choose, this expands to

$$\lambda f.\, \lambda x.\, (\lambda y.\, f\, (\lambda z.\, y\, y\, z))\, (\lambda y.\, f\, (\lambda z.\, y\, y\, z))\, x$$

Finally, we can η-convert to

$$\lambda f.\, (\lambda y.\, f\, (\lambda z.\, y\, y\, z))\, (\lambda y.\, f\, (\lambda z.\, y\, y\, z))$$

Voila!