How to Use the Y Cominbator

Chi Ian Tang

30 March 2022

---

1 Y Combinator

1.1 Definition

The Y Combinator is defined as:
$\mathbf{Y} \triangleq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

And it has the fixed-point property:
$\mathbf{Y}M =_\beta M(\mathbf{Y}M)$

This property is witnessed by the following β-reductions:

$$\begin{aligned}\mathbf{Y}M &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))M \\ &\to (\lambda x.M(xx))(\lambda x.M(xx)) \\ &\to M((\lambda x.M(xx))(\lambda x.M(xx)))\end{aligned}$$

$$\begin{aligned}M(\mathbf{Y}M) &= M((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))M) \\ &\to M((\lambda x.M(xx))(\lambda x.M(xx)))\end{aligned}$$

$$\begin{aligned}&\mathbf{Y}M \twoheadrightarrow M((\lambda x.M(xx))(\lambda x.M(xx))) \twoheadleftarrow M(\mathbf{Y}M) \\ &\mathbf{Y}M =_\beta M(\mathbf{Y}M)\end{aligned}$$

1.2 Recursive functions

Let's consider a simple template for recursive functions:

$$f(x) = \begin{cases} b(x) & \text{if } v(x) \\ g(x, f(h(x))) & \text{otherwise} \end{cases} \tag{1}$$

where g, h, v, b are some other functions, loosely representing the operation for aggregating the results (g), the operation to obtain the next recursion value (h), the verifying function which decides whether to continue the recursion (v), and the base case function (b). This will be referred to as the recursive function template (Eq. 1).
When we expand the function definition layer-by-layer (assuming $v(x)$ does not evaluates to true), we have:

$$\begin{aligned}f(x) &= g(x, f(h(x)) \\ &= g(x, g(h(x), f(h(h(x))))) \\ &= g(x, g(h(x), g(h(h(x)), f(h(h(h(x))))))) \\ &\dots\end{aligned}$$

In pseudo-code, we can implement the function as:

Listing 1: Pseudo-code Template for Recursive Functions (Template 1)

```
func f(x) {
    if v(x) {
        return b(x)
    } else {
        return g(x, f(h(x)))
    }
}
```

This will be referred to as the pseudo-code template for recursive functions.
A simple example of recursive function which fits the template is the factorial function:

$$\text{fact}(x) = \begin{cases} 1 & \text{if } x = 0 \\ x \times \text{fact}(x-1) & \text{otherwise} \end{cases}$$

We have the following mapping to the functions defined in the recursive function template (Eq. 1):

$$\begin{aligned} f &= \text{fact} \\ b(x) &= 1 \\ v(x) &= (x = 0) \\ h(x) &= x - 1 \\ g(x, r) &= x \times r \end{aligned}$$

This fits into our pseudo-code template as well:

```
func fact(x) {
    if x = 0 {
        return 1
    } else {
        return x * fact(x-1)
    }
}
```

1.3 β-conversion

Let's compare that to how the Y combinator undergoes β-conversion (non-normal order):

$$\begin{aligned} \mathbf{Y}F &=_\beta F(\mathbf{Y}F) \\ &=_\beta F(F(\mathbf{Y}F)) \\ &=_\beta F(F(F(\mathbf{Y}F))) \end{aligned}$$

We can see that it allows some form of recursion in λ-calculus.

But how do we make use of it to implement more familiar forms of recursion? How can we map recursive functions to λ-terms with the Y combinator?

1.4 Using the Y Combinator

The trick in using the Y combinator, is to define the accompanying λ-term (F) as a λ-abstraction with two or more variables, with the first acting like the recursive function that we are trying to define:

$$F \triangleq \lambda f \vec{x}.M$$

However, this is still very abstract, and it is more useful by defining the λ-term in the following form:

$$F \triangleq \lambda f \vec{x}.\mathbf{If}(V\vec{x})(B\vec{x})(G\vec{x}(f(H\vec{x})))$$

(see appendix for the definition of **If** and other common λ-terms)

To see how this helps us in defining recursive functions, let's look at the β-conversion of the Y combinator again, and this time with F defined in this specific form and an argument (assuming that $V\vec{x}$ does not evaluates to **True**):

$$\begin{aligned}
\mathbf{Y}F\vec{x} &=_\beta F(\mathbf{Y}F)\vec{x} \\
&=_\beta \mathbf{If}(V\vec{x})(B\vec{x})(G\vec{x}(\mathbf{Y}F(H\vec{x}))) \\
&=_\beta G\vec{x}(\mathbf{Y}F(H\vec{x})) \\
&=_\beta G\vec{x}(F(\mathbf{Y}F)(H\vec{x})) \\
&=_\beta G\vec{x}(G(H(\vec{x}))(\mathbf{Y}F(H(H\vec{x})))) \\
&=_\beta G\vec{x}(G(H(\vec{x}))(G(H(H(\vec{x})))(\mathbf{Y}F(H(H(H\vec{x})))))
\end{aligned}$$

You will notice, thanks to the fixed-point property of **Y**, this recursion matches exactly our template for recursive functions (Eq. 1) and the pseudo-code template (Template 1) in section 1.2. This means that we can define recursive functions in λ-calculus. In particular, any function that fits the recursive function template or the pseudo-code template can be defined in λ-calculus, by defining the accompanying λ-term in this specific form.

In the following sections, we will explore examples of how this knowledge can be used.

1.5 Primitive Recursion in λ-Calculus

Primitive recursion is defined in the following way [1]:

$$\rho(f,g)(\vec{x},x) \triangleq h(\vec{x},x) = \begin{cases} f(\vec{x}) & \text{if } x = 0 \\ g(\vec{x}, x-1, h(\vec{x}, x-1)) & \text{otherwise} \end{cases}$$

This is an instance of the recursive function template (Eq. 1) (using variables $\vec{x}, x$ rather than just x), once we substitute

h	for	f
$f(\vec{x})$	for	$b(\vec{x}, x)$
$(x = 0)$	for	$v(\vec{x}, x)$
$(\vec{x}, x - 1)$	for	$h(\vec{x}, x)$
$g(\vec{x}, x - 1, r)$	for	$g(\vec{x}, x, r)$

And the pseudo-code template (Template 1):

```
func h(x⃗, x) {
    if x = 0 {
        return f(x⃗)
    } else {
        return g(x⃗,x-1, h(x⃗, x-1))
    }
}
```

By assuming the existence of λ-terms $\textbf{If}, \textbf{Eq}_\textbf{0}, \textbf{Pred}$ (see appendix), which performs conditional branching, checks if some λ-term is 0, and calculates the predecessor (subtract by 1 if larger than 0), and F, G which represents functions f, g, we can define primitive recursion as $\lambda-$terms.

To see how to do this, we first define a λ-term in the specific form:

$$\mathbf{\Phi}_{f,g} \triangleq \lambda h \vec{x} x.\textbf{If}(V' \vec{x} x)(B' \vec{x} x)(G' \vec{x} x(h(H' \vec{x} x)))$$

We then define each of the components as follows:

$V' \vec{x} x = \textbf{Eq}_\textbf{0}\ x$	$(x = 0)$
$B' \vec{x} x = F\ \vec{x}$	$f(\vec{x})$
$H' \vec{x} x = (\vec{x}\ (\textbf{Pred}\ x))$	$(\vec{x}, x - 1)$
$G' \vec{x} x r = G\ \vec{x}\ (\textbf{Pred}\ x)\ r$	$g(\vec{x}, x - 1, r)$

We can then have primitive recursion represented as:

$$\mathbf{Y}\mathbf{\Phi}_{f,g} = \mathbf{Y}(\lambda h \vec{x} x.\textbf{If}\ (\textbf{Eq}_\textbf{0}\ x)\ (F\ \vec{x})\ (G\ \vec{x}\ (\textbf{Pred}\ x)(h\ \vec{x}\ (\textbf{Pred}\ x))))$$

To clearly point out how this is going to perform primitive recursion as we have intended, recall

$$F \triangleq \lambda f \vec{x}.\textbf{If}(V \vec{x})(B \vec{x})(G \vec{x}(f(H \vec{x})))$$
$$\mathbf{Y} F \vec{x} =_\beta G \vec{x}(G(H(\vec{x}))(G(H(H(\vec{x})))(\mathbf{Y} F(H(H(H \vec{x})))))$$

(under certain assumptions)
We can clearly see that the repeated application of terms G and H achieves the looping (recursive) behavior that we are looking for: at every iteration, we first check if x is zero, return $f(\vec{x})$ if yes, otherwise, we decrement x, call $g(\vec{x}, x - 1, h(\vec{x}, x - 1))$.

1.6 Minimisation in λ-Calculus

Minimization is defined in the following way:

$$\mu f(\vec{x}) \triangleq \text{the least } x \text{ such that } f(\vec{x}, x) = 0 \text{ and } \forall n < x. f(\vec{x}, n) \downarrow \text{ and } f(\vec{x}, n) \neq 0$$

From another angle, we can also define it as:

$$g(\vec{x}, x) \triangleq \begin{cases} x & \text{if } f(\vec{x}, x) = 0 \\ g(\vec{x}, x+1) & \text{otherwise} \end{cases}$$
$$\mu f(\vec{x}) \triangleq g(\vec{x}, 0)$$

This allows us to easily define a pseudo-code implementation (Template 1):

```
func g(x⃗, x) {
    if f(x⃗, x) = 0 {
        return x
    } else {
        return g(x⃗, x+1)
    }
}
func m(x⃗) {
    return g(x⃗, 0)
}
```

and this fits the recursive function template (Eq. 1) by substituting:

g	for	f
x	for	$b(\vec{x}, x)$
$(f(\vec{x}, x) = 0)$	for	$v(\vec{x}, x)$
$(\vec{x}, x+1)$	for	$h(\vec{x}, x)$
r	for	$g(\vec{x}, x, r)$

Note that the auxiliary function g is recursive, and matches our templates for using Y combinator. Therefore, we can define the function function g with the following λ-terms, by following a similar pattern that we have seen in primitive recursion:

$$\mathbf{\Psi}_f \triangleq \lambda g \vec{x} x. \mathbf{If} (V \vec{x} x)(B \vec{x} x)(G' \vec{x} x (g(H \vec{x} x)))$$

$V \vec{x} x = \mathbf{Eq_0}\ (F\ \vec{x}\ x)$	$(f(\vec{x}, x) = 0)$
$B \vec{x} x = x$	x
$H \vec{x} x = (\vec{x}\ (\mathbf{Succ}\ x))$	$(\vec{x}, x+1)$
$G' \vec{x} x r = r$	r

Therefore,

$$\begin{aligned}\mathbf{\Psi}_f &\triangleq \lambda g\vec{x}x.\mathbf{If}\ (\mathbf{Eq_0}\ (F\ \vec{x}\ x))\ x\ (g\ \vec{x}\ (\mathbf{Succ}\ x))\\ G &\triangleq \mathbf{Y}\Psi_f\end{aligned}$$

Finally, to define minimisation, we need to call function $g(\vec{x}, 0)$:

$$\begin{aligned}\mu f &\triangleq \lambda\vec{x}.G\ \vec{x}\ \underline{0}\\ &= \lambda\vec{x}.\mathbf{Y}\mathbf{\Psi}_f\ \vec{x}\ \underline{0}\\ &= \lambda\vec{x}.\mathbf{Y}(\lambda g\vec{x}x.\mathbf{If}\ (\mathbf{Eq_0}\ (F\ \vec{x}\ x))\ x\ (g\ \vec{x}\ (\mathbf{Succ}\ x)))\ \vec{x}\ \underline{0}\end{aligned}$$

1.7 Revisiting Factorial

We have seen how to define higher-order recursions in λ-calculus, but do we have a more concrete example?
Recall our earlier discussion on recursive functions (section 1.2): the factorial function is a good example of a recursive function. So, let's take that as an example.
Recall:

$$\text{fact}(x) = \begin{cases}1 & \text{if } x = 0\\ x \times \text{fact}(x-1) & \text{otherwise}\end{cases}$$

which matches the recursive function template (Eq. 1):

$$\begin{aligned}f &= \text{fact}\\ b(x) &= 1\\ v(x) &= (x = 0)\\ h(x) &= x - 1\\ g(x, r) &= x \times r\end{aligned}$$

and in pseudo-code (Template 1):

```
func fact(x) {
    if x = 0 {
        return 1
    } else {
        return x * fact(x-1)
    }
}
```

Again, we follow our previous examples, and we define

$$F \triangleq \lambda fx.\mathbf{If}(Vx)(Bx)(Gx(f(Hx)))$$

$$\begin{aligned} Vx &= \mathbf{Eq_0}\ x && (x = 0) \\ Bx &= \underline{1} && 1 \\ Hx &= \mathbf{Pred}\ x && x - 1 \\ Gxr &= \mathbf{Mult}\ x\ r && x \times r \end{aligned}$$

assuming we have λ-term **Mult**, which returns the Church encoding of the product of two numbers, and others defined.
Putting them together:

$$\mathbf{Fact} \triangleq \mathbf{Y}F \triangleq \mathbf{Y}(\lambda fx.\mathbf{If}\ (\mathbf{Eq_0}\ x)\ \underline{1}\ (\mathbf{Mult}\ x\ (f(\mathbf{Pred}\ x))))$$

This time, since we have every term defined, we can perform β-conversion on these λ-terms to verify the definition:

$$\begin{aligned} \mathbf{Fact}\ \underline{0} &= \mathbf{Y}F\ \underline{0} \\ &=_\beta F(\mathbf{Y}F)\ \underline{0} \\ &=_\beta (\lambda fx.\mathbf{If}\ (\mathbf{Eq_0}\ x)\ \underline{1}\ (\mathbf{Mult}\ x\ (f(\mathbf{Pred}\ x))))(\mathbf{Y}F)\ \underline{0} \\ &=_\beta \mathbf{If}\ (\mathbf{Eq_0}\ \underline{0})\ \underline{1}\ (\mathbf{Mult}\ \underline{0}\ ((\mathbf{Y}F)(\mathbf{Pred}\ \underline{0}))) \\ &=_\beta \underline{1} \\ &=_\beta \underline{\mathrm{fact}(0)} \end{aligned}$$

$$\begin{aligned} \mathbf{Fact}\ \underline{1} &= \mathbf{Y}F\ \underline{1} \\ &=_\beta F(\mathbf{Y}F)\ \underline{1} \\ &=_\beta \mathbf{If}\ (\mathbf{Eq_0}\ \underline{1})\ \underline{1}\ (\mathbf{Mult}\ \underline{1}\ ((\mathbf{Y}F)(\mathbf{Pred}\ \underline{1}))) \\ &=_\beta \mathbf{Mult}\ \underline{1}\ ((\mathbf{Y}F)(\mathbf{Pred}\ \underline{1})) \\ &=_\beta \mathbf{Mult}\ \underline{1}\ (\mathbf{Y}F\ \underline{0}) \\ &=_\beta \mathbf{Mult}\ \underline{1}\ \underline{1} \\ &=_\beta \underline{1} \\ &=_\beta \underline{\mathrm{fact}(1)} \end{aligned}$$

$$\begin{aligned} \mathbf{Fact}\ \underline{2} &= \mathbf{Y}F\ \underline{2} \\ &=_\beta F(\mathbf{Y}F)\ \underline{2} \\ &=_\beta \mathbf{If}\ (\mathbf{Eq_0}\ \underline{2})\ \underline{1}\ (\mathbf{Mult}\ \underline{2}\ ((\mathbf{Y}F)(\mathbf{Pred}\ \underline{2}))) \\ &=_\beta \mathbf{Mult}\ \underline{2}\ ((\mathbf{Y}F)(\mathbf{Pred}\ \underline{2})) \\ &=_\beta \mathbf{Mult}\ \underline{2}\ (\mathbf{Y}F\ \underline{1}) \\ &=_\beta \mathbf{Mult}\ \underline{2}\ \underline{1} \\ &=_\beta \underline{2} \\ &=_\beta \underline{\mathrm{fact}(2)} \end{aligned}$$

$$\begin{aligned}
\textbf{Fact}\ \underline{3} &= \mathbf{Y}F\ \underline{3} \\
&=_\beta F(\mathbf{Y}F)\ \underline{3} \\
&=_\beta \textbf{If}\ (\textbf{Eq}_\mathbf{0}\ \underline{3})\ \underline{1}\ (\textbf{Mult}\ \underline{3}\ ((\mathbf{Y}F)(\textbf{Pred}\ \underline{3}))) \\
&=_\beta \textbf{Mult}\ \underline{3}\ ((\mathbf{Y}F)(\textbf{Pred}\ \underline{3})) \\
&=_\beta \textbf{Mult}\ \underline{3}\ (\mathbf{Y}F\ \underline{2}) \\
&=_\beta \textbf{Mult}\ \underline{3}\ \underline{2} \\
&=_\beta \underline{6} \\
&=_\beta \underline{\text{fact}(3)}
\end{aligned}$$

By observing these conversions, you can see how the Y combinator works in real-time.

To properly show that **Fact** represents the factorial function, we prove it by mathematical induction:

Base case: $\textbf{Fact}\ \underline{0} =_\beta \underline{\text{fact}(0)}$ (shown above)
Inductive case: Assume $\textbf{Fact}\ \underline{k} = \mathbf{Y}F\ \underline{k} =_\beta \underline{\text{fact}(k)}$ for some $k \in \mathbb{N}$

$$\begin{aligned}
\textbf{Fact}\ \underline{k+1} &= \mathbf{Y}F\ \underline{k+1} \\
&=_\beta F(\mathbf{Y}F)\ \underline{k+1} && \text{fixed-point property of } \mathbf{Y} \\
&=_\beta \textbf{If}\ (\textbf{Eq}_\mathbf{0}\ \underline{k+1})\ \underline{1}\ (\textbf{Mult}\ \underline{k+1}\ ((\mathbf{Y}F)(\textbf{Pred}\ \underline{k+1}))) && \text{definition of } F \\
&=_\beta \textbf{Mult}\ \underline{k+1}\ ((\mathbf{Y}F)(\textbf{Pred}\ \underline{k+1})) && \text{definition of } \textbf{If}, \textbf{Eq}_\mathbf{0} \\
&=_\beta \textbf{Mult}\ \underline{k+1}\ (\mathbf{Y}F\ \underline{k}) && \text{definition of } \textbf{Pred} \\
&=_\beta \textbf{Mult}\ \underline{k+1}\ \underline{\text{fact}(k)} && \text{induction hypothesis} \\
&=_\beta \underline{(k+1) \times \text{fact}(k)} && \text{definition of } \textbf{Mult} \\
&=_\beta \underline{\text{fact}(k+1)} && \text{definition of fact}
\end{aligned}$$

Therefore, $\forall n \in \mathbb{N}.\textbf{Fact}\ \underline{n} =_\beta \underline{\text{fact}(n)}$

1.8 Ackermann Function

One of the most famous recursive functions is the Ackermann function [2], because of its rapid growth rate.
It is defined as:

$$\begin{aligned}
\text{ack}(0, y) &= y + 1 \\
\text{ack}(x + 1, 0) &= \text{ack}(x, 1) \\
\text{ack}(x + 1, y + 1) &= \text{ack}(x, \text{ack}(x + 1, y))
\end{aligned}$$

We can implement this as (Template 1):

```
func ack(x,y) {
    if x = 0 {
        return y + 1
    } else if y = 0 {
        return ack(x - 1, 1)
    } else {
        return ack(x - 1, ack(x, y - 1))
    }
}
```

This time, we have two conditional branches, and nested recursion. But as we see below, this does not prevent us from reusing the template that we have proposed above, by extending it:

$$A \triangleq \lambda fxy.\mathbf{If}(Vxy)(Bxy)(\mathbf{If}(V'xy)(B'fxy)(Gfxy))$$

$$\begin{aligned} Vxy &= \mathbf{Eq_0}\ x & (x = 0) \\ Bxy &= \mathbf{Succ}\ y & y + 1 \\ V'xy &= \mathbf{Eq_0}\ y & (y = 0) \\ B'fxy &= f\ (\mathbf{Pred}\ x)\ \underline{1} & \text{ack}(x-1, 1) \\ Gfxy &= f\ (\mathbf{Pred}\ x)\ (f\ x\ (\mathbf{Pred}\ y)) & \text{ack}(x-1, ack(x, y-1)) \end{aligned}$$

Note that we have added additional λ-terms to handle one more conditional branch, and modified the template slightly for B', G to let them be more general in order to handle the additional and nested recursions.
Putting them together:

$$\begin{aligned} &\mathbf{Ack} \triangleq \mathbf{Y}A \\ =&\mathbf{Y}(\lambda fxy.\mathbf{If}(\mathbf{Eq_0}\ x)(\mathbf{Succ}\ y)(\mathbf{If}(\mathbf{Eq_0}\ y)(f\ (\mathbf{Pred}\ x)\ \underline{1})(f\ (\mathbf{Pred}\ x)\ (f\ x\ (\mathbf{Pred}\ y)))))) \end{aligned}$$

We can show that **Ack** represents the Ackermann function by nested mathematical induction, i.e. $\forall x, y \in \mathbb{N}.\mathbf{Ack}\ \underline{x}\ \underline{y} =_\beta \underline{\text{ack}(x, y)}$:

Base case 1: $\forall y \in \mathbb{N}$

$$\begin{aligned} \mathbf{Ack}\ \underline{0}\ \underline{y} &= \mathbf{Y}A\ \underline{0}\ \underline{y} \\ &=_\beta A(\mathbf{Y}A)\ \underline{0}\ \underline{y} && \text{fixed-point property of } \mathbf{Y} \\ &=_\beta \mathbf{If}(\mathbf{Eq_0}\ \underline{0})(\mathbf{Succ}\ \underline{y})(\mathbf{If}(\mathbf{Eq_0}\ \underline{y})(\mathbf{Y}A\ (\mathbf{Pred}\ \underline{0})\ \underline{1})(\mathbf{Y}A\ (\mathbf{Pred}\ \underline{0})\ (\mathbf{Y}A\ \underline{0}\ (\mathbf{Pred}\ \underline{y})))) && \text{definition of } A \\ &=_\beta \mathbf{Succ}\ \underline{y} && \text{definition of } \mathbf{If}, \mathbf{Eq_0} \\ &=_\beta \underline{y+1} && \text{definition of } \mathbf{Succ} \\ &=_\beta \underline{\text{ack}(0, y)} && \text{definition of ack} \end{aligned}$$

Induction Hypothesis 1: Assume for some $x \in \mathbb{N}, \forall y \in \mathbb{N}.\mathbf{Ack}\ \underline{x}\ \underline{y} = \mathbf{Y}A\ \underline{x}\ \underline{y} =_\beta \underline{\text{ack}(x,y)}$
Base case 2, consider $x+1$:

$$
\begin{aligned}
&\mathbf{Ack}\ \underline{x+1}\ \underline{0} \\
&= \mathbf{Y}A\ \underline{x+1}\ \underline{0} \\
&=_\beta A(\mathbf{Y}A)\ \underline{x+1}\ \underline{0} && \text{property of } \mathbf{Y} \\
&=_\beta \mathbf{If}(\mathbf{Eq_0}\ \underline{x+1})(\mathbf{Succ}\ \underline{0})(\mathbf{If}(\mathbf{Eq_0}\ \underline{0})(\mathbf{Y}A\ (\mathbf{Pred}\ \underline{x+1})\ \underline{1})(\mathbf{Y}A\ (\mathbf{Pred}\ \underline{x+1})\ (\mathbf{Y}A\ \underline{x+1}\ (\mathbf{Pred}\ \underline{0})))) && \text{definition of } A \\
&=_\beta \mathbf{If}(\mathbf{Eq_0}\ \underline{0})(\mathbf{Y}A\ (\mathbf{Pred}\ \underline{x+1})\ \underline{1})(\mathbf{Y}A\ (\mathbf{Pred}\ \underline{x+1})\ (\mathbf{Y}A\ \underline{x+1}\ (\mathbf{Pred}\ \underline{0}))) && \text{definition of } \mathbf{If}, \mathbf{Eq_0} \\
&=_\beta \mathbf{Y}A\ (\mathbf{Pred}\ \underline{x+1})\ \underline{1} && \text{definition of } \mathbf{If}, \mathbf{Eq_0} \\
&=_\beta \mathbf{Y}A\ \underline{x}\ \underline{1} && \text{definition of } \mathbf{Pred} \\
&=_\beta \underline{\text{ack}(x,1)} && \text{Induction Hypothesis 1} \\
&=_\beta \underline{\text{ack}(x+1,0)} && \text{definition of ack}
\end{aligned}
$$

Induction Hypothesis 2: Assume for some $y \in \mathbb{N}.\mathbf{Ack}\ \underline{x+1}\ \underline{y} = \mathbf{Y}A\ \underline{x+1}\ \underline{y} =_\beta \underline{\text{ack}(x+1,y)}$
Inductive case, consider $x+1, y+1$:

$$
\begin{aligned}
&\mathbf{Ack}\ \underline{x+1}\ \underline{y+1} \\
&=\mathbf{Y}A\ \underline{x+1}\ \underline{y+1} \\
&=_\beta A(\mathbf{Y}A)\ \underline{x+1}\ \underline{y+1} && \text{property of } \mathbf{Y} \\
&=_\beta \mathbf{If}(\mathbf{Eq_0}\ \underline{x+1})(\mathbf{Succ}\ \underline{y+1})(\mathbf{If}(\mathbf{Eq_0}\ \underline{y+1}) \\
&\quad (\mathbf{Y}A\ (\mathbf{Pred}\ \underline{x+1})\ \underline{1})(\mathbf{Y}A\ (\mathbf{Pred}\ \underline{x+1})\ (\mathbf{Y}A\ \underline{x+1}\ (\mathbf{Pred}\ \underline{y+1})))) && \text{definition of } A \\
&=_\beta \mathbf{If}(\mathbf{Eq_0}\ \underline{y+1})(\mathbf{Y}A\ (\mathbf{Pred}\ \underline{x+1})\ \underline{1})(\mathbf{Y}A\ (\mathbf{Pred}\ \underline{x+1})\ (\mathbf{Y}A\ \underline{x+1}\ (\mathbf{Pred}\ \underline{y+1}))) && \text{definition of } \mathbf{If}, \mathbf{Eq_0} \\
&=_\beta \mathbf{Y}A\ (\mathbf{Pred}\ \underline{x+1})\ (\mathbf{Y}A\ \underline{x+1}\ (\mathbf{Pred}\ \underline{y+1})) && \text{definition of } \mathbf{If}, \mathbf{Eq_0} \\
&=_\beta \mathbf{Y}A\ \underline{x}\ (\mathbf{Y}A\ \underline{x+1}\ (\mathbf{Pred}\ \underline{y+1})) && \text{definition of } \mathbf{Pred} \\
&=_\beta \mathbf{Y}A\ \underline{x}\ (\mathbf{Y}A\ \underline{x+1}\ \underline{y}) && \text{definition of } \mathbf{Pred} \\
&=_\beta \mathbf{Y}A\ \underline{x}\ \underline{\text{ack}(x+1,y)} && \text{Induction Hypothesis 2} \\
&=_\beta \underline{\text{ack}(x,\text{ack}(x+1,y)} && \text{Induction Hypothesis 1} \\
&=_\beta \underline{\text{ack}(x+1,y+1)} && \text{definition of ack}
\end{aligned}
$$

Therefore, $\forall x, y \in \mathbb{N}.\mathbf{Ack}\ \underline{x}\ \underline{y} =_\beta \underline{\text{ack}(x,y)}$
You can see that since we define our λ-term by following the definition closely, the proof for correct representation of the Ackermann function is straight-forward.

(As an interesting side note, there exists an λ-term which represents the Ackermann function without involving the Y combinator: $\mathbf{Ack'} \triangleq \lambda x.\ x(\lambda f\ y.y\ f(f\ \underline{1}))\mathbf{Succ}$)

1.9 Summary

After we have seen all these examples of using the Y Combinator, we should be able to recognise functions that can be defined in λ-calculus.
The following gives a correspondence between different templates that we have used throughout this set of notes:

Recursive Function	Pseudo-code	λ-calculus
$f(x) = \begin{cases} b(x) & \text{if } v(x) \\ g(x, f(h(x))) & \text{otherwise} \end{cases}$	`func f(x) { if v(x) { return b(x) } else { return g(x, f(h(x))) } }`	$F \triangleq \mathbf{Y}(\lambda f x.\mathbf{If}(Vx)(Bx)(Gx(f(Hx))))$

1.10 Author's Comments

I hope that this set of notes helps clear up the mystery and potential confusion about the Y combinator. However, this is only one of many possible interpretations and uses of the Y combinator, and the discussion is surrounding the conversion of recursive functions of a particular form to λ-terms.

This was written based on the teaching materials of the 2021-2022 Computation Theory course (`https://www.cl.cam.ac.uk/teaching/2122/CompTheory/`) taught by Prof Andrew Pitts for the Part IB Computer Science Tripos at the University of Cambridge, and with the intention to complement the materials. However, I hope this can complement any other teaching materials on this topic.

I am supervising Part IB students for this course in 2021-2022.

1.11 Disclaimer

References

[1] Stephen Cole Kleene. General recursive functions of natural numbers. *Mathematische annalen*, 112(1):727–742, 1936.

[2] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99(1):118–133, 1928.

A Common λ-terms

$$\underline{n} \triangleq \lambda\ f\ x\ .\ f^n\ x$$
$$\textbf{If} \triangleq \lambda\ b\ x\ y\ .\ b\ x\ y$$
$$\textbf{True} \triangleq \lambda\ x\ y\ .\ x$$
$$\textbf{False} \triangleq \lambda\ x\ y\ .\ y$$
$$\textbf{Eq}_\textbf{0} \triangleq \lambda\ x\ .x(\lambda y.\textbf{False})\textbf{True}$$
$$\textbf{Pair} \triangleq \lambda\ x\ y\ f\ .\ f\ x\ y$$
$$\textbf{Fst} \triangleq \lambda\ p.p\ \textbf{True}$$
$$\textbf{Snd} \triangleq \lambda\ p.p\ \textbf{False}$$
$$\textbf{PredInc} \triangleq \lambda\ f\ p.\ \textbf{Pair}\ (f(\textbf{Fst}\ p))(\textbf{Fst}\ p)$$
$$\textbf{Pred} \triangleq \lambda\ n\ f\ x.\ \textbf{Snd}\ (n\ (\textbf{PredInc}\ f))(\textbf{Pair}\ n\ n))$$
$$\textbf{Mult} \triangleq \lambda\ m\ n\ f\ x.\ m\ (n\ f)\ x$$