isZero

Test an expression to see if it is c_0 or not: find arguments (for numerals) which yield `tru` if no successors have been applied, and `fls` otherwise.
i.e. $\texttt{iszero} = \lambda m.?$.

- c_0 returns its second argument, make it `tru` will yield $\texttt{iszro}\ c_0 = \texttt{tru}$
- All other numerals (where we want to return `fls`) applies s at least once!
- Make $s = \lambda x.\texttt{fls}$, ignoring its argument.

Putting that together:

$$\texttt{iszero} = \lambda m.m\,(\lambda x.\texttt{fls})\,\texttt{tru} \tag{1}$$

Testing it

$$\begin{array}{rl}
& \underline{\texttt{iszero}\, c_0} \\
& (\lambda m. m\, (\lambda x. \texttt{fls})\, \texttt{tru})\, c_0 \\
\to & c_0\, (\lambda x. \texttt{fls})\, \texttt{tru} \\
\to & (\lambda s. \lambda z.\, z)\, (\lambda x. \texttt{fls})\, \texttt{tru} \\
\to & (\lambda z.\, z)\, \texttt{tru} \\
\to & \texttt{tru} \\
\not\to &
\end{array}$$

The Mask of Zero

$$\underline{\texttt{iszero}\; c_2}$$

$$\begin{array}{ll}
 & (\lambda m. m\,(\lambda x.\texttt{fls})\,\texttt{tru})\; c_2 \\
\to & c_2\,(\lambda x.\texttt{fls})\,\texttt{tru} \\
\to & (\lambda s.\lambda z.\, s\,(s\; z))\,(\lambda x.\texttt{fls})\,\texttt{tru} \\
\to & (\lambda z.\,(\lambda x.\texttt{fls})\,((\lambda x.\texttt{fls})\; z))\,\texttt{tru} \\
\to & (\lambda x.\texttt{fls})\,((\lambda x.\texttt{fls})\,\texttt{tru}) \\
\to & \texttt{fls} \\
\not\to &
\end{array}$$

Predecessor(!)

Testing to see if something is zero is relatively straightforward, but predecessor requires some cleverness.

- In UAE, we defined `pred` as an annihilation operation over successors.
- In λ-Calculus, we essentially need to *reconstruct our numeral*, while keeping a *history of the previous value*.

$$\texttt{prd} = \lambda m.fst\,(m\,ss\,zz) \tag{2}$$

Where

$$ss = \lambda p.\texttt{pair}\,(\texttt{snd}\,p)\,(\texttt{plus}\,c_1\,(\texttt{snd p})) \tag{3}$$

$$zz = \texttt{pair}\,c_0\,c_0 \tag{4}$$

Converting back and forth:

$$\texttt{realbool} = \lambda b.b\ \texttt{true}\ \texttt{false} \tag{5}$$

$$\texttt{churchbool} = \lambda b.\texttt{if}\ \ b\ \texttt{then}\ \texttt{tru}\ \texttt{else}\ \texttt{fls} \tag{6}$$

$$\texttt{realnat} = \lambda c_n.c_n\,(\lambda x.\texttt{succ}\,x)\,0 \tag{7}$$

$$\texttt{churchnat} = \lambda n.\lambda s.\lambda z.\texttt{applyN}\ n\ s\ z \tag{8}$$

Curious Constructions

Consider the Ω-Function:

$$\Omega = (\lambda x.x\,x)(\lambda x.x\,x) \tag{9}$$

When you β-reduce Ω, you get Ω right back again!

$$(\lambda x.x\,x)(\lambda x.x\,x) \to (\lambda x.x\,x)(\lambda x.x\,x) \tag{10}$$

Because these functions do not converge to a normal form in a finite number of steps, they are known as **divergent**.

Y-Combinator

- The `Y-Combinator` *encodes* general recursion in the λ-Calculus.

$$\mathtt{Y} = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \tag{11}$$

- Unfortunately, it only works under call by name. The following **fixed-point combinator** solves the problem of general recursion for the call by value evaluation strategy.

$$\mathtt{fix} = \lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y)) \tag{12}$$

Factorial

The factorial function:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases} \tag{13}$$

We can encode it as follows:

$$g = \lambda \texttt{fct}.\lambda n.\texttt{if}\ \ n == 0\ \texttt{then}\ 1\ \texttt{else}\ \ n \times (\texttt{fct}\,(n-1)) \tag{14}$$

$$\texttt{factorial} = \texttt{fix}\ g \tag{15}$$

To save time and energy, we are encoding this using the enriched calculus.

Inductive Syntax of λ-Calculus

Let $\mathcal{V}$ be a countable set of variable names. The set of terms is the smallest set $\mathcal{T}$ such that:

1. $\mathcal{V} \subseteq \mathcal{T}$
2. $t_1 \in \mathcal{T} \wedge x \in \mathcal{V} \implies \lambda x.t_1 \in \mathcal{T}$
3. $t_1, t_2 \in \mathcal{T} \implies t_1\ t_2 \in \mathcal{T}$

- Via this definition, we can define size and depth the same way as we did under UAE.

Free Variables

The set of *free variables* of a term t, written $FV(t)$ is defined as follows:

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda x.t_1) & = & FV(t_1) \setminus \{x\} \\
FV(t_1 t_2) & = & FV(t_1) \cup FV(t_2)
\end{array}
$$

Substitution

The intuitive (but *wrong*) definition:

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } x \neq y \\
[x \mapsto s]\lambda y.t_1 & = & \lambda y.[x \mapsto s]t_1 & \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\,([x \mapsto s]t_2) &
\end{array}
$$

Why wrong?

This works reasonably well in most situations, such as the following:

$$[x \mapsto (\lambda z.z\ w)](\lambda y.x) \to \lambda y.\lambda z.z\ w \tag{16}$$

Consider the following:

$$[x \mapsto y](\lambda x.x) \to \lambda x.y \tag{17}$$

- This happens because we pass the substitution through lambdas without checking first to see if the variable we're replacing is bound!

Another try

If we fix the bit where we ignore bound vs. free variables...

$$
\begin{array}{lcll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} & = & \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{y} \neq \mathtt{x} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) & = & \begin{cases} \lambda\mathtt{y}.\ \mathtt{t}_1 & \\ \lambda\mathtt{y}.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 & \end{cases} & \begin{array}{l} \text{if } \mathtt{y} = \mathtt{x} \\ \text{if } \mathtt{y} \neq \mathtt{x} \end{array} \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) & = & ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

This expression now evaluates the way we expect it to...

$$[x \mapsto y](\lambda x.x) \to \lambda x.x \tag{18}$$

But the following expression doesn't.

$$[x \mapsto z](\lambda z.x) \to \lambda z.z \tag{19}$$

- When we sub in z, it becomes bound to λz.
- This is known as **variable capture**.

Accept No Substitutes!

In order to avoid having our variables captured, we might add the condition that, in order for a substitution to pass through a λ abstraction, the abstracted variable must not be in the set of free variables contained within the expression we are subbing in.

$$
\begin{array}{lcll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} & = & \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{y} \neq \mathtt{x} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{y}.\mathtt{t}_1) & = & \begin{cases} \lambda \mathtt{y}.\ \mathtt{t}_1 & \\ \lambda \mathtt{y}.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 & \end{cases} & \begin{array}{l} \text{if } \mathtt{y} = \mathtt{x} \\ \text{if } \mathtt{y} \neq \mathtt{x} \text{ and } \mathtt{y} \notin FV(\mathtt{s}) \end{array} \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) & = & ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

Still wrong

Consider the following example:

$$[x \mapsto y\ z](\lambda y.x\ y) \tag{20}$$

- No substitution can be performed, even though it would be reasonable to expect one.
- By relabelling y to some other arbitrary label, we can avoid the capture as well. For example:

$$[x \mapsto y\ z](\lambda y.x\ y) \to [x \mapsto y\ z](\lambda w.x\ w) \to (\lambda w.y\ z\ w) \tag{21}$$

Relabelling

By convention in λ-Calculus, terms that differ only in the names of bound variables are interchangeable in all contexts.
This is known as α-equivalence.
By working up to α, we can simplify our rules for substitution:

$$
\begin{array}{lcll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} & = & \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} & = & \mathtt{y} & \text{if } \mathtt{y} \neq \mathtt{x} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{y}.\mathtt{t}_1) & = & \lambda \mathtt{y}.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 & \text{if } \mathtt{y} \neq \mathtt{x} \text{ and } \mathtt{y} \notin FV(\mathtt{s}) \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) & = & [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2 &
\end{array}
$$

Operational Semantics of λ-Calculus

Here is the operational semantics of the CbV (call by value) λ-Calculus

$\to$ *(untyped)*

Syntax

$\mathtt{t}$ ::=		*terms:*
	$\mathtt{x}$	*variable*
	$\lambda\mathtt{x.t}$	*abstraction*
	$\mathtt{t\ t}$	*application*
$\mathtt{v}$ ::=		*values:*
	$\lambda\mathtt{x.t}$	*abstraction value*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-App2)}$$

$$(\lambda\mathtt{x.t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-AppAbs)}$$

Note that these are the semantics for the **pure λ-Calculus**.

Things of note

- All lambda terms are values (and vice-versa)
- One application rule (E-AppAbs), and two *congruence* rules, (E-App1) and (E-App2).
- Note how the placement of values controls the flow of execution.
  - We may only proceed with (E-App2) if t_1 is a value, implying that (E-App1) is inapplicable.
  - The reason this strategy is called "call by value" is because the term being substituted in (E-AppAbs) must be a value.