COMPSCI 3MI3 - Principles of Programming Languages

Topic 6 - Untyped Lambda Calculus 2 : Extensions and Formalities

NCC Moore

McMaster University

Fall 2021

Adapted from "Types and Programming Languages" by Benjamin C. Pierce

More UAE Terms in λ Calculus

Over the course of this topic, we will continue to expand our knowledge of λ-Calculus.

- We will polish off the last remaining terms of UAE.
- We will discuss the **enrichment** of the calculus, so that it is less of a pain to work with.
- We will examine the ways in which λ-Calculus can be used as a general computational engine.
- We will discuss the formalities of the language.

isZero

In order to test a expression to see if it is c_0 or not, we must find arguments for the Church numerals which yield `tru` if no successors have been applied, and `fls` otherwise.

- Here's a reminder of what Church numerals look like:

$$c_0 = \lambda s.\lambda z.\, z \tag{1}$$
$$c_1 = \lambda s.\lambda z.\, s\ z \tag{2}$$
$$c_2 = \lambda s.\lambda z.\, s\,(s\ z) \tag{3}$$
$$c_3 = \lambda s.\lambda z.\, s\,(s\,(s\ z)) \tag{4}$$
$$\vdots$$

Coke Zero

To design a function that returns `tru` or `fls` when applied to Church numerals, we need to find some **inputs** to a church numeral which yield the correct **output**.

- c_0 simply echos the second argument, so making the second argument `tru` will yield `iszro` $c_0 =$ `tru`
- We might observe that each numeral that would return `fls` applies z to one or more s.
- So, if we make s something that always returns `fls`, no matter what is applied to it, we have success!
  - $\lambda x.$`fls` is the simplest function that fits the above description.

Zero to Hero

So, our `iszro` function needs to take a church numeral, and apply the above functions to it.

$$\texttt{iszero} = \lambda m.m : (\lambda x.\texttt{fls})\,\texttt{tru} \tag{5}$$

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

The Mask of Zero

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

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*.

$$\texttt{prd} = \lambda m.fst\,(m\;ss\;zz) \tag{6}$$

Where

$$ss = \lambda p.\texttt{pair}\,(\texttt{snd}\;p)\,(\texttt{plus}\;c_1\,(\texttt{snd}\;\texttt{p})) \tag{7}$$

$$zz = \texttt{pair}\;c_0\;c_0 \tag{8}$$

Predecessor of c_2

$$
\begin{array}{ll}
 & \underline{\texttt{pred}\ c_2} \\
 & (\lambda m. fst\ (m\ ss\ zz))\ c_2 \\
\rightarrow & fst\ (c_2\ ss\ zz) \\
\rightarrow & (\lambda p.\ p\ tru)\ (c_2\ ss\ zz) \\
\rightarrow & c_2\ ss\ zz\ tru \\
\rightarrow & (\lambda s. \lambda z.\ s\ (s\ z))\ ss\ zz\ tru \\
\rightarrow & (\lambda z.\ ss\ (ss\ z))\ zz\ tru \\
\rightarrow & ss\ (ss\ zz)\ tru \\
\rightarrow & (\lambda p. \texttt{pair}\ (\texttt{snd}\ p)\ (\texttt{plus}\ c_1\ (\texttt{snd}\ p)))\ (ss\ zz)\ \texttt{tru} \\
\rightarrow & \texttt{pair}\ (\texttt{snd}\ (ss\ zz))\ (\texttt{plus}\ c_1\ (\texttt{snd}\ (\texttt{ss zz})))\ \texttt{tru} \\
\rightarrow & (\lambda f. \lambda s. \lambda b.\ b\ f\ s)\ (\texttt{snd}\ (ss\ zz))\ (\texttt{plus}\ c_1\ (\texttt{snd}\ (\texttt{ss zz})))\ \texttt{tru} \\
\rightarrow\ \rightarrow\ \rightarrow & \texttt{tru}\ (\texttt{snd}\ (ss\ zz))\ (\texttt{plus}\ c_1\ (\texttt{snd}\ (\texttt{ss zz}))) \\
\rightarrow & (\lambda t. \lambda f. t)\ (\texttt{snd}\ (ss\ zz))\ (\texttt{plus}\ c_1\ (\texttt{snd}\ (\texttt{ss zz}))) \\
\rightarrow\ \rightarrow & \texttt{snd}\ (ss\ zz)
\end{array}
$$

Predecessor of c_2

$$
\begin{array}{rl}
 & \mathtt{snd}\,(ss\;zz) \\
\to & (\lambda p.\;p\,\mathtt{fls})\,(ss\;zz) \\
\to & ss\;zz\,\mathtt{fls} \\
\to & (\lambda p.\mathtt{pair}\,(\mathtt{snd}\;p)\,(\mathtt{plus}\;c_1\;(\mathtt{snd}\;\;\mathtt{p})))\;zz\,\mathtt{fls} \\
\to & \mathtt{pair}\,(\mathtt{snd}\;zz)\,(\mathtt{plus}\;c_1\;(\mathtt{snd}\;\;\mathtt{zz}))\,\mathtt{fls} \\
\to & (\lambda f.\lambda s.\lambda b.\;b\;f\;\;s)\,(\mathtt{snd}\;zz)\,(\mathtt{plus}\;c_1\;(\mathtt{snd}\;\;\mathtt{zz}))\,\mathtt{fls} \\
\to\to\to & \mathtt{fls}\,(\mathtt{snd}\;zz)\,(\mathtt{plus}\;c_1\;(\mathtt{snd}\;\;\mathtt{zz})) \\
\to\to & \mathtt{plus}\;c_1\;(\mathtt{snd}\;\;\mathtt{zz}) \\
\to & (\lambda m.\lambda n.\lambda s.\lambda z.\;m\;s\;(n\;s\;z))\;c_1\;(\mathtt{snd}\;\;\mathtt{zz}) \\
\to\to & \lambda s.\lambda z.\;c_1\;\;s\;((\mathtt{snd}\;\;\mathtt{zz})\;s\;z) \\
\not\to &
\end{array}
$$

Latent Evaluations

Because we are using the call by value strategy, we actually can't evaluate any further than this.

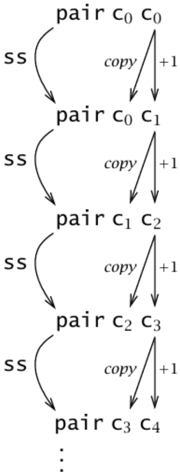
- Remember that call by value doesn't allow us to evaluate the subexpressions of a term with a lambda.
- This leads to an interesting property of λ-Calculus: **Latent Evaluations**.
- Undersupplied λ expressions won't fully execute, in effect, storing evaluations for later retrieval.
- These evaluations will only be calculated if the expression is supplied with a sufficient number of arguments.
- We often say these forms are **functionally equivalent** to the normal forms that would arise from full β-reduction, and use them interchangeably.

Continuing using normal order...

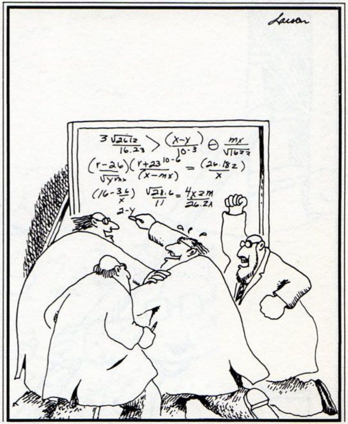
$$
\begin{array}{rl}
 & \lambda s.\lambda z.\, c_1\, s\, ((\texttt{snd}\ \texttt{zz})\, s\, z) \\
\rightarrow & \lambda s.\lambda z.\, (\lambda s.\lambda z.\, s\, z)\, s\, ((\texttt{snd}\ \texttt{zz})\, s\, z) \\
\rightarrow\rightarrow & \lambda s.\lambda z.\, s\, (\texttt{snd}\ \texttt{zz}\, s\, z) \\
\rightarrow & \lambda s.\lambda z.\, s\, ((\lambda p.\, p\, \texttt{fls})\, \texttt{zz}\, s\, z) \\
\rightarrow & \lambda s.\lambda z.\, s\, (\texttt{zz}\, \texttt{fls}\, s\, z) \\
\rightarrow & \lambda s.\lambda z.\, s\, (\texttt{pair}\, c_0\, c_0\, \texttt{fls}\, s\, z) \\
\rightarrow & \lambda s.\lambda z.\, s\, ((\lambda f.\lambda s.\lambda b.\, b\, f\, s)\, c_0\, c_0\, \texttt{fls}\, s\, z) \\
\rightarrow\rightarrow\rightarrow & \lambda s.\lambda z.\, s\, (\texttt{fls}\, c_0\, c_0\, s\, z) \\
\rightarrow\rightarrow & \lambda s.\lambda z.\, s\, (c_0\, s\, z) \\
\rightarrow & \lambda s.\lambda z.\, s\, (c_0\, s\, z) \\
\rightarrow & \lambda s.\lambda z.\, s\, ((\lambda s.\lambda z.\, z)\, s\, z) \\
\rightarrow\rightarrow & \lambda s.\lambda z.\, s\, z \\
\rightarrow & c_1 \\
\not\rightarrow &
\end{array}
$$

For higher numerals...

- The way this algorithm works is to start at c_0, and build our way up to the number we're trying to take the predecessor of.
- One half of the pair keeps track of the last numeral we were on, so when we reach the numeral we're trying to take the predecessor of...
- We just need to skim off the first element.



Enriching the λ-Calculus



"Go for it, Sidney! You've got it! You've got it! Good hands! Don't choke!"

That Was Pretty Painful!

As the previous, 3 full slide derivation has demonstrated, λ-Calculus can be pretty painful to do any actual work in!

- Wouldn't it be convenient if there was some sort of advanced electrical computational engine that could perform such calculations...
- Hundreds even, in the blink of an eye!

Hopefully we are now satisfied that all of our usual values and operators have some expression under λ-Calculus. For convenience of calculation, let's move away from the pure system, and add some additional semantic content.

Adding UAE

We can convert easily between terms in UAE and our previously defined λ expression equivalents:

$$\texttt{realbool} = \lambda b.b\ \texttt{true}\ \texttt{false} \tag{9}$$

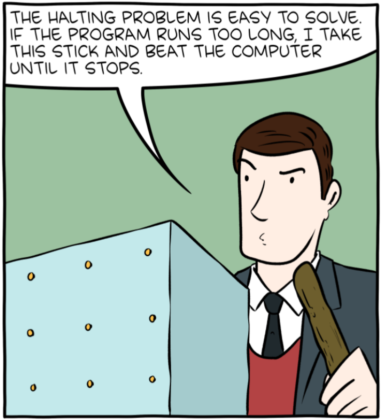
$$\texttt{churchbool} = \lambda b.\texttt{if}\ \ b\ \texttt{then}\ \texttt{tru}\ \texttt{else}\ \texttt{fls} \tag{10}$$

$$\texttt{realnat} = \lambda c_n.c_n\,(\lambda x.\texttt{succ}\,x)\,0 \tag{11}$$

$$\texttt{churchnat} = \lambda n.(\lambda \texttt{succ}.\lambda 0.n)\,s\,z \tag{12}$$

From here, it's just a matter of using the right operations on the right values.

Turing Completeness and Lambda Calculus



What if Alan Turing had been an engineer?

Turing Completeness/Equivalence

To review, a system of computation is considered **Turing Complete** or `Turing Equivalent` if it can perform all the actions of a Turing machine. The minimal set of things you need to be able to do is:

- Support conditional branching.
  - This implies support for arbitrary go-to operations.
- An infinite amount of tape (or memory).

Technically, no physical computer is Turing Complete, because of physical constraints on memory.

Church-Turing Thesis

The Church-Turing Thesis states:

- A function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine.
- A secondary effect of this is that a program is computable via a Turing Machine iff it is computable using a λ expression.

UAE is not Turing Complete because the Theorem of Evaluation holds.

- This theorem means a UAE expression's evaluation chain must be finite for a finite term.
- We can construct a finite Turing Machine which runs infinitely.

Curious Constructions

Theorem of Evaluation does not hold for λ-Calculus!

- This doesn't mean it isn't determinate, because it is (depending on your evaluation strategy).

This only means that finite λ expressions can have infinite evaluation chains, such as the Ω-Function:

$$\Omega = (\lambda x.x\,x)(\lambda x.x\,x) \tag{13}$$

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

$$(\lambda x.x\,x)(\lambda x.x\,x) \to (\lambda x.x\,x)(\lambda x.x\,x) \tag{14}$$

Because these functions do not converge on a normal form, they are known as **divergent**.

R(e(c(u(r(s(i(o(n())))))))))

The Ω-Function is an interesting function, but it isn't very practical.

- Its cousin, the `Y-Combinator`, encodes general recursion in the λ-Calculus.

$$\mathtt{Y} = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)) \tag{15}$$

- 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.

$$\mathtt{fix} = \lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y)) \tag{16}$$

Factorial Again

Recall the factorial function:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases} \tag{17}$$

We can now encode it as follows.

$$g = \lambda \mathtt{fct}.\lambda n.\mathtt{if}\ \ n == 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ n \times (\mathtt{fct}\,(n-1)) \tag{18}$$

$$\mathtt{factorial} = \mathtt{fix}\ \ g \tag{19}$$

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

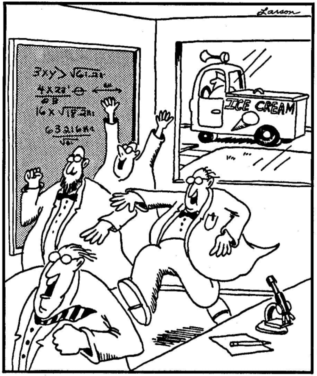
Example Time!

$$\underline{\texttt{factorial 3}}$$

$$\begin{array}{cl}
 & \texttt{fix}\, g\, 3 \\
\to & (\lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y)))\ g\ 3 \\
\to & (\lambda x.g\ (\lambda y.x\ x\ y))\ (\lambda x.g\ (\lambda y.x\ x\ y))\ 3 \\
\text{setting} & h = \lambda x.g\ (\lambda y.x\ x\ y) \\
\text{yields} & (\lambda x.g\ (\lambda y.x\ x\ y))\ h\ 3 \\
\to & g\ (\lambda y.h\ h\ y)\ 3 \\
\text{setting} & \texttt{fct} = \lambda y.h\ h\ y \\
\text{yields} & g\ \texttt{fct}\ 3 \\
\to & (\lambda \texttt{fct}.\lambda n.\texttt{if}\ \ n == 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ n \times (\texttt{fct}\ (n-1)))\ \texttt{fct}\ 3 \\
\to\ \to & \texttt{if}\ \ 3 == 0\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 3 \times (\texttt{fct}\ (3-1)) \\
\to\ \to & 3 \times (\texttt{fct}\ 2) \\
\to & 3 \times ((\lambda y.h\ h\ y)\ 2) \\
\to & 3 \times (h\ h\ 2)
\end{array}$$

$$
\begin{array}{rl}
 & 3 \times (h\,h\,2) \\
\to & 3 \times ((\lambda x.g\,(\lambda y.x\,x\,y))\,h\,2) \\
\to & 3 \times (g\,\mathtt{fct}\,2) \\
\to\to & 3 \times (g\,\mathtt{fct}\,2) \\
\to\to\to & 3 \times (\mathtt{if}\ \ 2 == 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ 2 \times (\mathtt{fct}\,(2-1))) \\
\to\to & 3 \times 2 \times (\mathtt{fct}\,1) \\
\to\to\to & 6 \times (h\,h\,1) \\
\to\to & 6 \times (g\,\mathtt{fct}\,1) \\
\to\to\to & 6 \times (\mathtt{if}\ \ 1 == 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ 1 \times (\mathtt{fct}\,(1-1))) \\
\to\to & 6 \times 1 \times (\mathtt{fct}\,0) \\
\to\to\to & 6 \times (h\,h\,0) \\
\to\to & 6 \times (g\,\mathtt{fct}\,0) \\
\to\to\to & 6 \times (\mathtt{if}\ \ 0 == 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ 0 \times (\mathtt{fct}\,(0-1))) \\
\to\to\to & 6 \\
\not\to &
\end{array}
$$

Formalities



"There may, indeed, be other applications of the system than its use as a logic."

- Alonzo Church, 1932 -

Inductive Syntax of λ-Calculus

For the rest of this topic, we will examine the subtleties of a more rigorous definition of the λ calculus, beginning with an inductive definition of its syntax.

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 with Every Order

We can define a new function over λ-Calculus, in the style of the `consts` operator of UAE.

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

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

Use Substitute!

At the beginning of our discussion of the λ-Calculus, we said it would be necessary to develop a semantic of both the calculus itself, and the substitution operation. Let's start with substitution.

- We will start with an intuitive definition based on our knowledge of elementary-school algebra, and develop a more robust definition by exposing issues with the naive approach.

Let us define naive substitution as follows:

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

Substitute is a Move in the Pokemon Games

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

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

But the naive description contains a bug!

- Consider the following:

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

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

Maybe I Should Have Substituted a Better Joke...

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...

$$[x \mapsto y](\lambda x.x) \to \lambda x.x \tag{22}$$

But the following expression doesn't.

$$[x \mapsto z](\lambda z.x) \to \lambda z.z \tag{23}$$

- 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.

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

No Substitutions, Extensions or Refunds!

We're not out of the woods yet, however.

- Consider the following example:

$$[x \mapsto y\ z](\lambda y.x\ y) \tag{24}$$

- 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:

$$[x \mapsto y\ z](\lambda y.x\ y) \to [x \mapsto y\ z](\lambda w.x\ w) \to (\lambda w.y\ z\ w) \tag{25}$$

Relabel Them Variables!

By convention in λ-Calculus, terms that differ only in the names of bound variables are interchangeable in all contexts.

- By adding the meta-rule that we rename variables whenever a substitution would result in variable capture, we can actually simplify our rules for substitution:

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

Operational Semantics of λ-Calculus

Finally, we are ready to discuss the operational semantics of the call by value evaluation strategy of λ-Calculus

$\rightarrow$ *(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

- The set of values here is somewhat more interesting than in UAE.
  - Since this strategy doesn't evaluate past λ's, all λ abstractions are values.
- In these semantics, we have 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.

λ-Calculus Self Interpreter

A self-interpreter is a program which implements its own semantics.

- Some programming languages, including Haskell, have their compilers and interpreters implemented in the language they are implementing.
  - Python's interpreter is written in C...

The following is a self-interpreting λ expression, reliant on the Y-Combinator.

$$Y\,(\lambda e.\lambda m.m\,(\lambda x.x)\,(\lambda m.\lambda n.e\ m\,(e\ n))\,(\lambda m.\lambda v.e\,(m\ v))) \tag{26}$$

I don't have an example of it's operation, I just found it while researching this slide deck, thought it was cool, and threw it in at the end.

- Mogensen, T. (1994). Efficient Self-Interpretation in Lambda Calculus. Journal of Functional Programming.

Last Slide Comic