COMPSCI 3MI3 - Principles of Programming Languages

Topic 5 - Untyped Lambda Calculus

NCC Moore

McMaster University

Fall 2021

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


THINK
LAMBDA
imgflip.com

Computation my Friends! Computation!

In the 1960s, Peter Landin observed that complex programming languages can be understood by capturing their essential mechanisms as a small core calculus.

- The core language used by Landin was λ-**Calculus**
  - Developed in the 1920s by Alonzo Church.
  - Reduces *all* computation to function **definition** and **application**.

The strength of λ-Calculus comes from it's *simplicity* and its capacity for **formal reasoning**.

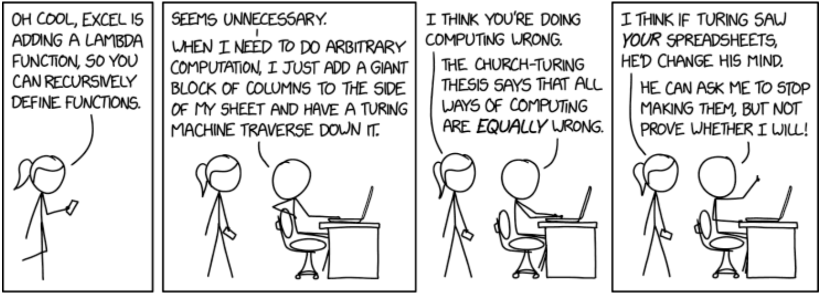
Languages Based on λ-Calculus

Haskell

OCaml

Lisp

Inclusion of lambda expressions in multi-paradigm languages has been a trend recently...



The Tactile (As Opposed to Visual) Basics

Abstraction is the programmer's most reliable weapon. For a true programmer, the following statement should be instinctively irritating:

```
(5*4*3*2*1) + (7*6*5*4*3*2*1) + (3*2*1)
```

Your humble professor couldn't even type the above without using copy-paste. Our instinct tells us to rewrite the above as:

```
factorial(n) = if n = 0 then 1 else n * factorial (n-1)
factorial(5) + factorial(7) + factorial(3)
```

Baby's First λ

```
factorial(n) = if n = 0 then 1 else n * factorial (n-1)
```

Right now, the left-hand side is doing too much work (that is, any at all). Let's introduce a new operator, λ, which does the work of (n) in the above.

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

In Haskell, this would be written:

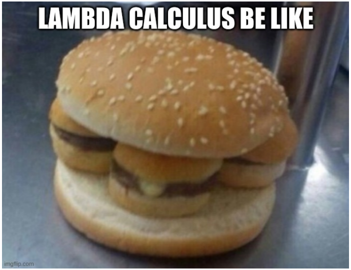
```
factorial =
  (\ n -> if n = 0 then 1 else (factorial (n-1)))
```

- In equation 1, **function application** is in the traditional $f(x)$ form, whereas Haskell and λ-Calculus use a space character for function application.

λ-Calculus

In λ-Calculus, *everything* is either a function definition or a function application of this form.

- The arguments accepted by functions are functions
- The results returned by functions are also functions.



λ-Calculus Syntax

Untyped λ-Calculus is comprised of only 3 terms!

$$\begin{aligned} \langle t \rangle \ ::= \ & \langle x \rangle \\ | \ & \lambda \langle x \rangle \ . \ \langle t \rangle \\ | \ & \langle t \rangle \ \langle t \rangle \end{aligned}$$

These terms are:

- Variable names
- λ Abstraction
- Function Application.


ONIONS HAVE LAYERS...
SYNTAX HAS LAYERS
imgflip.com

Layers of Syntax

When working with programming languages, it is useful to be able to re-organize, and even transform our syntax before applying semantics to it. It is very common to distinguish:

- **Concrete Syntax**
  - The syntax the programmer actually encodes the program in.
- from **Abstract Syntax**
  - A tree structure containing the terms of the program

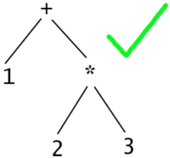
It is sometimes useful to specify even more layers than these.

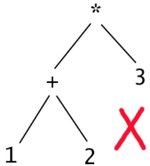
- The syntax of a complex language can often be vastly simplified via purely syntactic transformations.
- Redundant constructs can be simplified to reduce the number of distinct terms (aka **desugaring**)
  - e.g., changing array access to pointer arithmetic in C

AST

Abstract syntax is an excellent way of visualizing a program's structure, especially in resolving operator precedence.

- For example, under BEDMAS, the expression $1 + 2 * 3$ would be the left diagram, not the right diagram:



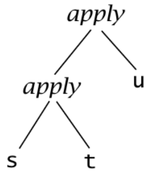


BEDMAS trees are evaluated leaf-first, but as we'll see, in λ expressions may be evaluated using a number of different strategies.

ASTs of λ-Calculus

In order to reduce the number of redundant parentheses in our concrete syntax for λ-Calculus:

- Function application will be **left-associative**. That is, `s t u` is interpretted as:



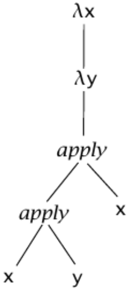
- or, in concrete syntax, `(s t) u`

Scope of λ Operator

The abstraction operator λ is taken to extend to the right as far as possible. For the following expression:

- λx.λy.x y x

We would construct an AST:



Free vs Bound Variables

In predicate calculus, we recognize a distinction between **free** and **bound** variables.

$$\exists x \mid x \neq y \tag{2}$$

In the above:

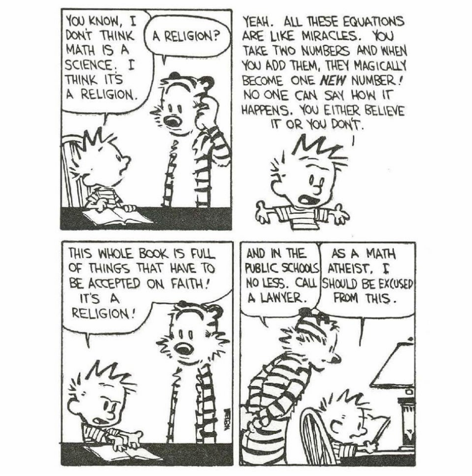
- x is **bound** by the existential quantifier.
- y is not bound by a quantifier and is therefore **free**

In *lambda*-Calculus, we apply the same concepts and the same terms to the relationship between variables and the abstraction operator λ.

$$(\lambda x.x\ y)\ x \tag{3}$$

- The first occurance of x is **bound**.
- Both y and the second occurrance of x are **free**.

Operational Semantics



Only One Evaluation Rule!

Each execution step performs a function application on a term with at least one abstracted variable.

- This means both a λ abstraction *and* a function application must be present and adjacent for a term to be reducible.

These terms reduce by **substituting** the abstracted variable with the term applied to the function. In other words:

$$(\lambda x.t_1)\ t_2 \to [x \mapsto t_2]\ t_1 \tag{4}$$

- A λ expression which may be simplified is known as a **redex**, or *reducible expression*.
- The above evaluation process is known as **beta-reduction**.

Using All our Substitutions

In the previous slide, the symbol $[x \mapsto t_2]\, t_1$ stands for "the term obtained by the replacement of all free occurances of x in t_1 by t_2.

- We will eventually need to define *two* sets of operational semantics, one for rewriting lambda expressions, and another for performing substitutions.

Examples:

$$(\lambda x.x)\, y \to y \tag{5}$$

$$(\lambda x.x\, (\lambda x.x))\, (u\, r) \to u\, r\, (\lambda x.x) \tag{6}$$

Note in this last example that the substitution operation does not pass to the inner λ expression. This is because occurances of x inside this expression are not **free**, but **bound** to the containing abstraction.

Evaluation Dilemma!

So far, we have a reasonably rigorous definiton for beta reduction, and our intuitions about substitution derived from our high-school algebra classes.

- The goal is be able to create an algorithm which evaluates lambda expressions.
- What happens if we have a choice of multiple beta-reductions in a single λ expression?
- We need an *evaluation strategy*, which we can build into our operational semantics.

Our Test Expression

To examine strategies, we will use a running example expression:

$$(\lambda x.x)\,((\lambda x.x)\,(\lambda z.(\lambda x.x)\,z)) \tag{7}$$

- $\lambda x.x$ is effectively an **identity function**, so we write it as id.

$$id\,(id\,(\lambda z.id\,z)) \tag{8}$$

The above expression has three redexes:

$$\color{red}{id\,(id\,(\lambda z.id\,z))} \tag{9}$$

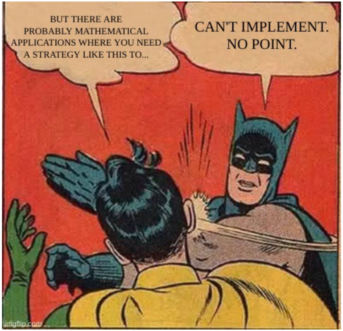
$$id\,(\color{red}{id\,(\lambda z.id\,z)}) \tag{10}$$

$$id\,(id\,(\lambda z.\color{red}{id\,z})) \tag{11}$$

The Worst Strategy Ever

Under **Full Beta-Reduction**, the redexes may be reduced in any order.

- Full beta-reduction is not really even a stategy.
- This algorithm is non-deterministic.



Normal Order

Normal order begins with the leftmost, outermost redex, and proceeds until there are no more redexes to evaluate. This is the way a human would probably choose to it if they weren't thinking about it too hard.

$$\begin{array}{ll} & id\ (id\ (\lambda z.id\ z)) \\ \rightarrow & id\ (\lambda z.id\ z) \\ \rightarrow & \lambda z.id\ z \\ \rightarrow & \lambda z.z \\ \nrightarrow & \end{array}$$

Under this strategy (and those to follow), evaluation is a *partial function*, as each term t evaluates to *at most* one term t'

Call By Name

The **call by name** strategy is more restrictive than normal order.
You can't evaluate anything that isn't an outer-most term.

$$
\begin{array}{rl}
 & id\ (id\ (\lambda z.id\ z)) \\
\to & id\ (\lambda z.id\ z) \\
\to & \lambda z.id\ z \\
\not\to &
\end{array}
$$

In this case, $\lambda z.id\ z$ is considered a **normal form**.

Haskell is Cool!

An optimized version of call by name strategy, called **call by need** is used by Haskell to evaluate expressions.

- In order to avoid having to re-evaluate the arguments of expressions, Haskell overwrites all occurances of an expression the first time that expression is evaluated.
- As a result, they only need to be evaluated *once*.
- Effectively, this is a reduction relation on syntax **graphs**, rather than syntax **trees**.

Call By Value

Most languages use **call by value**, where only the outermost redexes are reduced, and a redex is only reduced when the right-hand-side has already been reduced to a value.

- Here, as elsewhere, a value is a term in normal form.

$$
\begin{array}{ll}
 & id\ (id\ (\lambda z.id\ z)) \\
\to & id\ (\lambda z.id\ z) \\
\to & \lambda z.id\ z \\
\not\to &
\end{array}
$$

We will be using this strategy a lot, because it is commonly implemented in programming languages, and easier to enrich with added features.

Curry in a Hurry!

You may have noticed that so far our functions have only taken one argument.

- It would be almost trivial to define an extension to our calculus which allows multiple arguments.
- We don't have to, however, because of **currying**.

Because our functions are **higher order functions**, that is, they can return a function as their result, we can describe a function taking multiple arguments as a series of functions taking one argument, that pass their result to the next.

Coconut Lamb Curry!

This is how we might pass multiple arguments in a richer language:

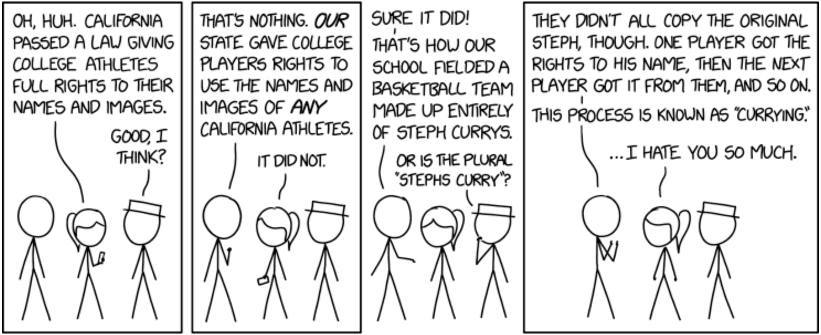
$$(\lambda(x, y, z).s)(a, b, c) \to [x \mapsto a][y \mapsto b][z \mapsto c]s \tag{12}$$

In our calculus, the following statement is equivalent.

$$\begin{aligned} & (\lambda x.\lambda y.\lambda z.\ s)\ a\ b\ c \\ \to\quad & (\lambda y.\lambda z.\ [x \mapsto a]s)\ b\ c \\ \to\quad & (\lambda z.\ [x \mapsto a][y \mapsto b]s)\ c \\ \to\quad & [x \mapsto a][y \mapsto b][z \mapsto c]s) \\ \not\to\quad & \end{aligned}$$

It should be noted that our untyped λ-Calculus has not been designed for use by programmers, but for greater simplicity in proving mathematical properties.

Church Booleans



Strong Like Bool

Now that we have our new mode of computation established, let's start reconstructing the elements of UAE, starting with the Booleans!

- We can define true and false values as follows.
  - We use `tru` and `fls` here to avoid confusion with the `true` and `false` of UAE.

$$\texttt{tru} = \lambda t.\lambda f.t \tag{13}$$

$$\texttt{fls} = \lambda t.\lambda f.f \tag{14}$$

Wait, What?

Our definitions of Boolean values won't make a lot of sense until we show how they're used. Consider the following λ expression, reproducing UAE's `if then else` term:

$$\texttt{test} = \lambda t_1.\lambda t_2.\lambda t_3.\ t_1\ t_2\ t_3 \tag{15}$$

With $t_1 = \texttt{tru}$	With $t_1 = \texttt{fls}$
$(\lambda t_1.\lambda t_2.\lambda t_3.\ t_1\ t_2\ t_3)\ \texttt{tru}\ u\ v$	$(\lambda t_1.\lambda t_2.\lambda t_3.\ t_1\ t_2\ t_3)\ \texttt{fls}\ u\ v$
$\to (\lambda t_2.\lambda t_3.\ \texttt{tru}\ t_2\ t_3)\ u\ v$	$\to (\lambda t_2.\lambda t_3.\ \texttt{fls}\ t_2\ t_3)\ u\ v$
$\to (\lambda t_3.\ \texttt{tru}\ u\ t_3)\ v$	$\to (\lambda t_3.\ \texttt{fls}\ u\ t_3)\ v$
$\to \texttt{tru}\ u\ v$	$\to \texttt{fls}\ u\ v$
$\to (\lambda t.\lambda f.t)\ u\ v$	$\to (\lambda t.\lambda f.f)\ u\ v$
$\to (\lambda f.u)\ v$	$\to (\lambda f.f)\ v$
$\to u$	$\to v$
$\not\to$	$\not\to$

Boolean Operators

Adding pieces to λ-Calculus is very different from adding pieces to UAE.

- To expand UAE, we needed to add additional terms and evaluation rules.
  - The more terms and evaluation rules we add, the longer and more complicated our proofs become!
- By contrast, when we "added" Booleans to our λ-Calculus, *nothing actually had to be added to the language itself!*
  - `tru` and `fls` are not terms, but **labels** for λ expressions *that were already valid terms!*

Conservative Extension

Consider two theories, T_1 and T_2. We say that T_2 is a **conservative extension** of T_1 if:

- Every theorem of T_1 is a theorem of T_2
- Any theorem of T_2 in the language of T_1 is already a theorem of T_1.

The relationship between λ-Calculus, and λ-Calculus with Booleans fits the above description.

- We did not have to introduce any additional theorems to describe the Booleans.
- Our Boolean extension still has all the rules of λ-Calculus.

The reason this is useful, is that anything proven about λ-Calculus is *automatically true* of any conservative extension!

Boolean And

Since adding language elements is so easy under λ-Calculus, let's add a few more!

$$\text{and} = \lambda b.\lambda c.\ b\ c\ \mathit{fls} \tag{16}$$

$$\begin{array}{ll} & \underline{\text{With input } \texttt{tru}\ \texttt{tru}} \\ & (\lambda b.\lambda c.\ b\ c\ \texttt{fls})\ \texttt{tru}\ \texttt{tru} \\ \rightarrow & (\lambda c.\ \texttt{tru}\ c\ \texttt{fls})\ \texttt{tru} \\ \rightarrow & \texttt{tru}\ \texttt{tru}\ \texttt{fls} \\ \rightarrow & (\lambda t.\lambda f.t)\ \texttt{tru}\ \texttt{fls} \\ \rightarrow & (\lambda f.\texttt{tru})\ \texttt{fls} \\ \rightarrow & \texttt{tru} \\ \not\rightarrow & \end{array}$$

$$\begin{array}{ll} & \underline{\text{With input } \texttt{tru}\ \texttt{fls}} \\ & (\lambda b.\lambda c.\ b\ c\ \texttt{fls})\ \texttt{tru}\ \texttt{fls} \\ \rightarrow & (\lambda c.\ \texttt{tru}\ c\ \texttt{fls})\ \texttt{fls} \\ \rightarrow & \texttt{tru}\ \texttt{fls}\ \texttt{fls} \\ \rightarrow & (\lambda t.\lambda f.t)\ \texttt{fls}\ \texttt{fls} \\ \rightarrow & (\lambda f.\texttt{fls})\ \texttt{fls} \\ \rightarrow & \texttt{fls} \\ \not\rightarrow & \end{array}$$

Completing Our Truth Table

With input $\texttt{fls tru}$

$$
\begin{array}{ll}
 & (\lambda b.\lambda c.\ b\ c\ \texttt{fls})\ \texttt{fls}\ \texttt{tru} \\
\to & (\lambda c.\ \texttt{fls}\ c\ \texttt{fls})\ \texttt{tru} \\
\to & \texttt{fls}\ \texttt{tru}\ \texttt{fls} \\
\to & (\lambda t.\lambda f.f)\ \texttt{tru}\ \texttt{fls} \\
\to & (\lambda f.f)\ \texttt{fls} \\
\to & \texttt{fls} \\
\not\to &
\end{array}
$$

With input $\texttt{fls fls}$

$$
\begin{array}{ll}
 & (\lambda b.\lambda c.\ b\ c\ \texttt{fls})\ \texttt{fls}\ \texttt{fls} \\
\to & (\lambda c.\ \texttt{fls}\ c\ \texttt{fls})\ \texttt{fls} \\
\to & \texttt{fls}\ \texttt{fls}\ \texttt{fls} \\
\to & (\lambda t.\lambda f.f)\ \texttt{fls}\ \texttt{fls} \\
\to & (\lambda f.f)\ \texttt{fls} \\
\to & \texttt{fls} \\
\not\to &
\end{array}
$$

Pairs

Using the selectivity of Church Booleans, we can easily use them to encode **pairs**.

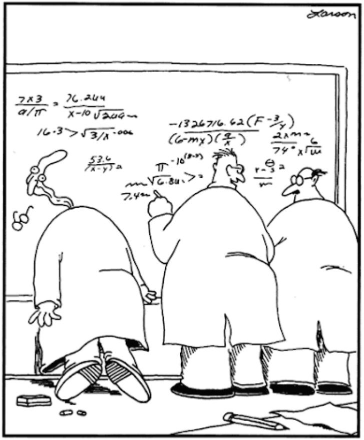
$$\texttt{pair} = \lambda f.\lambda s.\lambda b.\ b\ f\ s \tag{17}$$

$$\texttt{fst} = \lambda p.\ p\ tru \tag{18}$$

$$\texttt{snd} = \lambda p.\ p\ fls \tag{19}$$

- b is used to select between f and s
- `fst` and `snd` merely apply `tru` and `fls` respectively.
- Since `tru` selects the first argument, it also selects the first term in the pair.
- Likewise for `fls`

Church Numerals



“Ha! Webster’s blown his cerebral cortex.”

Church Numerals

We can define the natural numbers in λ-Calculus in a manner still quite similar to Peano arithmetic.

$$c_0 = \lambda s.\lambda z.\, z \tag{20}$$
$$c_1 = \lambda s.\lambda z.\, s\, z \tag{21}$$
$$c_2 = \lambda s.\lambda z.\, s\,(s\, z) \tag{22}$$
$$c_3 = \lambda s.\lambda z.\, s\,(s\,(s\, z)) \tag{23}$$
$$\vdots$$

In other words, Church numerals take two arguments, a successor s and a zero term z.

- z is applied to s, and the result is applied to another s and so on, until we reach n applications.

Correspondance with Booleans

The observant student may have noticed that c_0 has the same definition as `fls`.

- This is sometimes called a **pun** in computer science.
- The same thing occurs in lower level languages, where the interpretation of a sequence of bits is context dependant.
- In C, the bit arrangement 0x00000000 corresponds to:
  - Zero (Integer)
  - False (Boolean)
  - "\0\0\0\0" (Character Array)

Succ-ess!

We can define the successor function on Church Numerals as follows:

$$\texttt{succ} = \lambda n.\lambda s.\lambda z.\ s\,(n\ s\ z) \tag{24}$$

Successor of Two

$$\begin{array}{ll}
 & \texttt{succ}\ \ c_2 \\
\rightarrow & (\lambda n.\lambda s.\lambda z.\ s\,(n\ s\ z))\ c_2 \\
\rightarrow & \lambda s.\lambda z.\ s\,(c_2\ s\ z) \\
\rightarrow & \lambda s.\lambda z.\ s\,((\lambda s.\lambda z.\ s\,(s\ z))\ s\ z) \\
\rightarrow & \lambda s.\lambda z.\ s\,((\lambda z.\ s\,(s\ z))\ z) \\
\rightarrow & \lambda s.\lambda z.\ s\,(s\,(s\ z)) \\
\rightarrow & c_3 \\
\not\rightarrow &
\end{array}$$

Add-itional Functions!

Similarly, we can define addition as follows:

$$\texttt{plus} = \lambda m.\lambda n.\lambda s.\lambda z.\ m\, s(n\, s\, z) \tag{25}$$

Freedom is the freedom to say...

$$\begin{array}{ll}
 & \texttt{plus}\ c_2\ c_2 \\
\to & (\lambda m.\lambda n.\lambda s.\lambda z.\ m\ s\ (n\ s\ z))c_2 c_2 \\
\to & (\lambda n.\lambda s.\lambda z.\ c_2\ s\ (n\ s\ z))c_2 \\
\to & \lambda s.\lambda z.\ c_2\ s\ (c_2\ s\ z) \\
\to & \lambda s.\lambda z.\ (\lambda s.\lambda z.\ s\ (s\ z))\ s\ ((\lambda s.\lambda z.\ s\ (s\ z))\ s\ z) \\
\to & \lambda s.\lambda z.\ (\lambda z.\ s\ (s\ z))\ ((\lambda s.\lambda z.\ s\ (s\ z))\ s\ z) \\
\to & \lambda s.\lambda z.\ (s\ (s\ ((\lambda s.\lambda z.\ s\ (s\ z))\ s\ z))) \\
\to & \lambda s.\lambda z.\ (s\ (s\ ((\lambda z.\ s\ (s\ z))\ z))) \\
\to & \lambda s.\lambda z.\ (s\ (s\ (s\ (s\ z)))) \\
\to & c_4 \\
\not\to &
\end{array}$$

Times Have Changed

Finally, let's define a multiplication operator.

$$times = \lambda m.\lambda n.\ m\,(\texttt{plus}\ n)\ c_0 \tag{26}$$

$$\underline{3 \times 2 =?}$$

$$\begin{array}{ll} & \texttt{times}\ c_3\ c_2 \\ \to & (\lambda m.\lambda n.\ m\,(\texttt{plus}\ n)\ c_0)\ c_3\ c_2 \\ \to & (\lambda n.\ c_3\,(\texttt{plus}\ n)\ c_0)\ c_2 \\ \to & (\lambda s.\lambda z.\ s\,(s\,(s\ z)))\,(\texttt{plus}\ c_2)\ c_0 \\ \to & (\texttt{plus}\ c_2)\,((\texttt{plus}\ c_2)\,((\texttt{plus}\ c_2)\ c_0)) \end{array}$$

Sub-Derivation

Technically this is cheating, since we don't have a rule for this type of substitution in the semantic, and it violates our evaluation strategy.

$$
\begin{array}{ll}
 & \texttt{plus}\ c_2 \\
\rightarrow & (\lambda m.\lambda n.\lambda s.\lambda z.\ m\ s\ (n\ s\ z))\ (\lambda s.\lambda z.\ s\ (s\ z)) \\
\rightarrow & (\lambda n.\lambda s.\lambda z.\ (\lambda s.\lambda z.\ s\ (s\ z))\ s\ (n\ s\ z)) \\
\rightarrow & (\lambda n.\lambda s.\lambda z.\ (\lambda z.\ s\ (s\ z))\ (n\ s\ z)) \\
\rightarrow & (\lambda n.\lambda s.\lambda z.\ (s\ (s\ (n\ s\ z))))
\end{array}
$$

(It saves a lot of time though)

$$
\begin{array}{ll}
 & (\texttt{plus}\ c_2)\ ((\texttt{plus}\ c_2)\ ((\texttt{plus}\ c_2)\ c_0)) \\
\rightsquigarrow & (\lambda n.\lambda s.\lambda z.\ (s\ (s\ (n\ s\ z))))\ ((\texttt{plus}\ c_2)\ ((\texttt{plus}\ c_2)\ c_0)) \\
\rightarrow & \lambda s.\lambda z.\ (s\ (s\ (((\texttt{plus}\ c_2)\ ((\texttt{plus}\ c_2)\ c_0))\ s\ z))) \\
\rightsquigarrow & \lambda s.\lambda z.\ (s\ (s\ (((\lambda n.\lambda s.\lambda z.\ (s\ (s\ (n\ s\ z))))\ ((\texttt{plus}\ c_2)\ c_0))\ s\ z))) \\
\rightarrow & \lambda s.\lambda z.\ (s\ (s\ ((\lambda z.\ (s\ (s\ (((\texttt{plus}\ c_2)\ c_0)\ s\ z))))\ z))) \\
\rightarrow & \lambda s.\lambda z.\ (s\ (s\ (s\ (s\ (((\texttt{plus}\ c_2)\ c_0)\ s\ z))))) \\
\rightsquigarrow & \lambda s.\lambda z.\ (s\ (s\ (s\ (s\ (((\lambda n.\lambda s.\lambda z.\ (s\ (s\ (n\ s\ z))))\ c_0)\ s\ z))))) \\
\rightarrow & \lambda s.\lambda z.\ (s\ (s\ (s\ (s\ ((\lambda s.\lambda z.\ (s\ (s\ (c_0\ s\ z))))\ s\ z))))) \\
\rightarrow & \lambda s.\lambda z.\ (s\ (s\ (s\ (s\ ((\lambda z.\ (s\ (s\ (c_0\ s\ z))))\ z))))) \\
\rightarrow & \lambda s.\lambda z.\ (s\ (s\ (s\ (s\ (s\ (s\ (c_0\ s\ z))))))) \\
\rightarrow & \lambda s.\lambda z.\ (s\ (s\ (s\ (s\ (s\ (s\ ((\lambda s.\lambda z.\ z)\ s\ z))))))) \\
\rightarrow & \lambda s.\lambda z.\ (s\ (s\ (s\ (s\ (s\ (s\ ((\lambda z.\ z)\ z))))))) \\
\rightarrow & \lambda s.\lambda z.\ (s\ (s\ (s\ (s\ (s\ (s\ z)))))) \\
\not\rightarrow &
\end{array}
$$

Last Slide Comic