λ^nulus

My intepretation of the Lambda Calculus derivative that we have produced. In this document all variables are assumed to have unique names. Fuck you De-Brujin indices. This contains an untyped implementation to clarify syntax and semantics. A typed implementation will follow when we finalise the intepretation.

1 Syntax

Given by the below grammar

$$\begin{array}{lcl} e & \rightarrow & \lambda\{\, x_1 \ , \ldots \ , x_n \,\}\, .\, M \ \mid \ (e_1 \ @p \ e_2) \mid \ x \\ p & \rightarrow & [x_1 \ , \ldots \ , x_n] \\ x & \rightarrow & \mathit{String} \end{array}$$

The first e production rules represents an abstraction of n variables within M. This represents *all* the bound variables within M. (I feel like this could potentially be extended to also include the standard Lambda Calculus app to allow for nested Lambdas but don't really see what that would buy you).

A key point that I haven't quite figured out is what is M. M appears to be some kind of λ^n-expression but with no free variables.

The second e production rule represents the delayed application of another expression to a given variable found at the path represented by p. A path is a list of variables that show where within and to what variable the tree of suspended computations to apply a given expression. There exist various constraints on paths i.e. that they represent valid paths down the computational tree.

Examples of this calculus can be found below:

$$(((\lambda\{x_1, x_2\}\, .\, x_1 + x_2) \ @[x_1] \ \ 3) \ @[x_2] \ \ 3) \tag{1}$$

$$(((\lambda\{x_1, x_2\}\,.\,x_1 + x_2)\ \ @[x_1]\ \ (\lambda\{y_1, y_2\}\,.\,y_1 + y_2))\ \ @[x_1, y_1]\ \ 3) \tag{2}$$

As far as I can tell there exists 1 or potentially 2 reduction rules on this calculus. An @ production can be reduced as follows (This is equivalent to flattening a block):

$$((\lambda\{x_1, x_2\}\,.\,x_1 + x_2)\ \ @[x_1]\ \ (\lambda\{y_1, y_2\}\,.\,y_1 + y_2)) \tag{3}$$

$$= ((\lambda\{y_1, y_2, x_2\}\,.\,y_1 + y_2 + x_2) \quad | \ @ - equivalence \tag{4}$$

In our implementation this reduction applies recursively to any λs nested deeper than our original λ

There also exists an operation to reduce any λ^n-expression to a λ-expression. This involves applying @-equivalence until all $@p$ are consumed. Once this has been done it can be reduced to a canonical form.

Given that every λ^n-expression is equivalent to $|\texttt{capturing\_set}|!$ equivalent λ-expressions in the ordinary λ Calculus we can abitrarily produce a λ-expression as follows:

$$\lambda\{x_1, ..., x_n\}\,.\,M \tag{5}$$

$$= \lambda x_1. \ ... \ \lambda x_n\,.\,M \tag{6}$$

The above operation can be used alongside the path to target a specific region of blocks. The second operation needs to be followed by some sort of lift operation. (This seems like it could be categorical) This needs to be investigated more as there are terms which don't map to the λ^n-Calculus as nicely as it would be hoped.

The lift operator essentially delves into the structure of the λ-expression and pulls all binders to the outermost level. As follows:

$$\lambda\{x_1, ..., x_n\}\,.\,M \tag{7}$$

$$= \lambda x_1. \ ... \ \lambda x_n\,.\,M \tag{8}$$

This calculus could potentially be extended so that we can apply a list of expressions at a given path producing the production rules as follows

$$
\begin{array}{lcl}
e & \rightarrow & \lambda\{\, x_1 \ , \ldots \ , x_n \,\}\, .\, M \quad | \quad (e \ \ @p \ \ [\, e_1 \ , \ldots \ , \, e_n]) \,| \quad x \\
p & \rightarrow & [x_1 \ , \ldots \ , [y_1 \ , \ldots \ , y_n] \ \ @x_n] \\
x & \rightarrow & \mathit{String}
\end{array}
$$

This would allow the binding of any number of the variables in the receiving expression.

I envision this could be used as follows:

$$\lambda\{x_1, x_2, x_3\}\, .\, x_1 + x_2 + x_3 \quad @[[x_1, x_2]] \quad [3, 3] \tag{9}$$

Any expression in this second calculus is equivalent to a number of expressions of the first calculus.

You could probably keep going with the extended version so that you could bind to abitrary many variables with the same expression but this is all I can be bothered to type up for now.