An Interactive Lambda Calculus Interpreter and Visualization Tool

Bhashithe Abeysinghe[1], Murray Patterson[2], and Rajshekhar Sunderraman[2(✉)]

[1] American Institutes for Research, Arlington, VA, USA
[2] Georgia State University, Atlanta, GA, USA
{mpatterson30,rsunderraman}@gsu.edu

Abstract. Lambda Calculus forms the foundational basis for all functional programming languages. In addition, since most other modern programming languages are slowly incorporating functional features such as higher-order functions, lambda expressions, and combinatory logic operators, one can argue that a basic understanding of Lambda Calculus is necessary for modern computer scientists. Typically, Lambda Calculus is introduced in the undergraduate Computer Science curriculum in the Programming Languages Concepts, Programming Paradigms, or Functional Programming courses. In this paper, we present a command-line tool and an interactive Web visualization tool that provide a platform for the student to learn to formulate and evaluate lambda expressions. In the command-line tool, the user can compute free-variables in lambda expressions, perform alpha-reductions and substitutions, and simplify lambda expressions using beta-reductions. This interface is convenient for evaluating lambda expressions to its most reduced form by going through a series of beta-reductions. In the visualization tool, the user inputs a lambda expression and interactively evaluates the expressions, one beta-reduction at a time. The lambda expression is shown in the form of an expression tree, with ready to execute function application nodes shown in green. The student can click on any of these green nodes to perform a beta reduction and the resulting tree is shown. The student can also go back to any of the previous trees to try other beta reductions. The tool was validated in an offering of the Programming Language Concepts class and results from the study are presented in the paper.

Keywords: Lambda Calculus · Visualization · Interactive Tool

1 Introduction

Lambda Calculus [1,13] is a formal model of computation based on two simple ideas: function abstraction and function application. Introduced by Alonzo Church in the early 1930s, this model of computation forms the basis for modern day functional programming languages such as Haskell, OCaml, etc. Lambda Calculus has been proven to be equivalent to the other popular model of Computation, the Turing Machine [12], which forms the basis for modern procedural

A. Karkare et al. (Eds.): COMPUTE 2024, CCIS 2400, pp. 113–125, 2025.
https://doi.org/10.1007/978-3-031-84391-4_9

or state-based programming languages. Procedural or state-based programming languages, such as Python, Java, or Javascript, incorporate many ideas from functional programming, including lambda expressions and higher-order functions. For these reasons, it is important for a Computer Science student to be well versed in the basics of the Lambda Calculus.

Lambda Calculus is usually introduced in a Programming Language Concepts course or in a course on Functional Programming in the undergraduate curriculum, or in a theoretical course on Models of Computation. Lambda Calculus concepts include the (simple) syntax of function abstractions and function applications, and semantics that include concepts such as free and bound variables, α-reductions, substitutions, and β-reductions.

Even though the syntax of Lambda expressions is quite simple, with just two operators, the nested structure of these expressions causes a lot of difficulty in understanding their structure. Having a tool that can automatically produce a visual expression tree for a given Lambda expression will enable students to better understand the nested structures in expressions. Further, the semantics of Lambda Calculus is defined in terms of β-reductions, a transformation that is mathematically defined and which operates on the tree structure of the expressions, thereby causing considerable difficulty for the student in understanding the semantics. Incorporating user interactivity in the visual tool to be able to produce the result of a β-reduction on an existing tree to visualize the next tree in the evaluation can go a long way in improving the student's understanding of semantics. We present such an interactive visual tool that accepts a Lambda expression as input and displays a visual expression tree with cues to apply β-reductions. The user is provided with various options to simplify the expression and can see the workings of the evaluation process. They can also get an understanding of the recursive structures in these expressions. A command line version of the tool is also available and complements the functionalities of the visual tool. The command line version allows users to "execute" lambda expressions all the way to termination, thereby allowing students to test their ability to compose programs from expressions.

The remainder of this paper is organized as follows: Sect. 2 covers basics of Lambda Calculus, including syntax, expression trees, and semantics. Section 3 discusses the command line tool and Sect. 4 describes the interactive visual tool. Section 5 describes an evaluation study of the tool, as well as the results and possible improvements. Section 6 summarizes related work, while Sect. 7 concludes the paper and suggests future work.

2 Lambda Calculus

Lambda Calculus was introduced by Church in the 1930s as a model of computation. The general idea was to define computation as consisting of two main constructs: function definition and function application. This model of computation forms the foundation of modern day functional programming languages such as Haskell, etc.

2.1 Syntax

Definition 1. *A λ-expression is defined inductively as follows:*

1. *A variable is a λ-expression (e.g. x, y, m, n, etc.).*
2. *If M is a λ-expression and x is a variable, then (λ x.M) is a λ-expression.*
3. *If M and N are λ-expressions then (M N) is a λ-expression.*
4. *A number is a λ-expression (e.g.* 20*,* 7.5*,* −2.22*, etc.).*
5. *If M and N are λ-expressions and op is an arithmetic operator such as* +*,* −*,* ∗*, or* /*, then (op M N) is a λ-expression.*

In the above definition,

- $(\lambda x.M)$ is called a lambda abstraction; or in programming terminology the definition of a function. Here, x is the input parameter (bound variable) and M is the body of the function. Note that all function definitions in Lambda Calculus take exactly one input parameter.
- $(M\ N)$ is called a function application; or in programming terminology a function call. M is the function being called and N is the actual argument provided as input to M.
- The number and arithmetic operator λ-terms (4. and 5., respectively) are not part of the original Lambda Calculus and are introduced to make it interesting while using the interactive tool.

Here are some examples of λ-expressions:

1. (((λ x. (λ y. (+ x y))) 10) 5)
2. (((λ f. (λ x. (f x))) (λ y. (* y y))) 12)
3. (λ x. (λ y. (λ z. ((x z)(y z)))))
4. ((((λ f. (λ x. ((f x) f))) (λ y. (λ g. (g (* y y))))) 2) (λ a. a))

2.2 Expression Trees

Lambda expressions can be visualized as an expression tree with interior nodes denoting lambda abstractions, function applications, or arithmetic operators and leaves denoting variables or numbers. An example expression tree is shown in Fig. 1 using a screenshot from our system.

2.3 Semantics

The semantics of lambda expressions is defined through a series of formal definitions that follow. To start with, we denote the **scope** of parameter variable x in (λ x.M) to be the body M. The following defines the concept of bound and free variables.

Definition 2. *In the λ-expression (λx.M), the variable occurrence immediately after λ is said to be* **bound***. In addition, any occurrence of x in M that is not otherwise* **bound** *to another λ-expression within M is said to be* **bound** *to the parameter variable x in (λ x.M). Any variable that is not bound is said to be* **free***.*

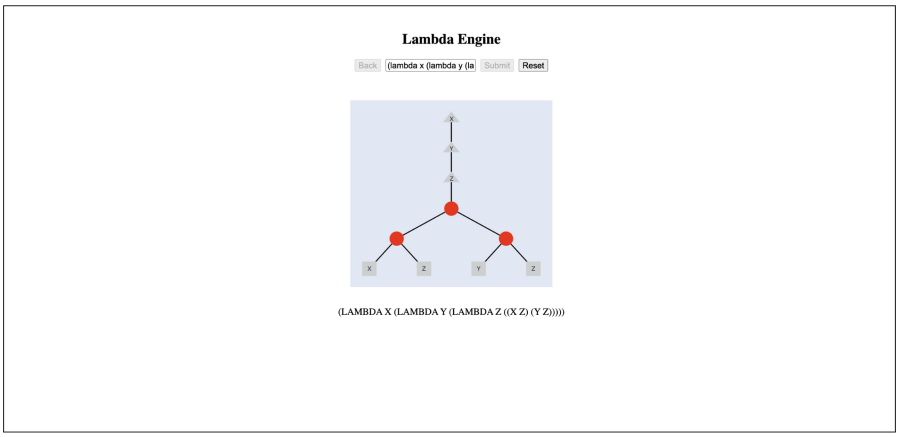

Fig. 1. Expression tree for (λ x.(λ y.(λ z.((x z) (y z))))))

Definition 3. *The set of free variables in M, denoted by FV[M], is inductively defined as follows:*

1. $FV[x] = x$
2. $FV[(\lambda\ x.M)] = FV[M] \setminus \{ x \}$
3. $FV[(M\ N)] = FV[M] \cup FV[N]$
4. $FV[number] = \{ \}$
5. $FV[(op\ M\ N)] = FV[M] \cup FV[N]$

The next concept is that of α-equivalence. Two function abstractions are α-equivalent if the two differ only in the name of the formal parameter/input. This relates to the naming of the formal parameter. For example, the λ-expressions (λ x.x) and (λ y.y) are essentially the same. Renaming the bound variable does not change the function abstraction. More formally,

Definition 4. *$(\lambda\ x.M) =_\alpha (\lambda\ y.M\ \{x \leftarrow y\})$ where y is a "brand new" variable not appearing in M, and $M\ \{x \leftarrow y\}$ is M with all occurrences of x replaced by y.*

The next concept needed in defining the semantics is that of substitutions. A substitution replaces a free variable in a λ-expression with another λ-expression. Formally, substitutions are defined as follows:

Definition 5. *Substitute free variable x in a lambda expression by another lambda expression P:*
$x\ [x = P] = P$
$y\ [x = P] = y$, *if* $x \neq y$
$(M\ N)\ [x = P] = (M[x = P]\ N[x = P])$
$(\lambda\ x.M)\ [x = P] = (\lambda\ x.M)$
$(\lambda\ y.M)\ [x = P] = (\lambda\ y.M[x = P])$, *if* $x \neq y$ *and* $y \notin FV[P]$

(λ y.M) [x = P] = (λ y'.(M {y ← y'}[x = P])), if x ≠ y and y ∈ FV[P] and y' is brand new

Some examples are shown below:
(λ x. (x y)) [y = 5] = (λ x. (x 5))
(λ x. (x y)) [y = (u v)] = (λ x. (x (u v)))

Substitution must be done carefully so as not to alter the meaning of the λ-expression, for example,
(λ x. (x y)) [y = x] ≠ (λ x. (x x))

As can be seen, y was a free-variable before, but after the substitution y's value has become bound. Such a case is called a "capture" case, and a way around changing the meaning of the substitutions is first to recognize the fact that we have a "capture" case and then rename the parameter variable in the λ-expression (using an α-equivalence) with a brand new variable and then perform the substitution. So,
(λ x.(x y)) [y = x] $=_\alpha$ (λ x'.(x' y)) [y = x] = (λ x'.(x' x))

The final concept is that of β-reduction, defined as follows:

Definition 6. *((λ x.M) N) $=_\beta$ M[x = N]*

As an example, consider the λ-expression, (λ x. (* x x)), that denotes the "square" function. To call this function with argument 5, we will construct the "apply" λ-term: ((λ x. (* x x)) 5). β-reduction allows us to "execute or call" a function. We "substitute" the bound variable (parameter), x, of the function abstraction with 5 in the body of the function abstraction.

((λ x. (* x x)) 5) = (* x x) [x = 5] = (* 5 5) = 25

β–reduction can be applied only to a λ-expression of the form ((λ x.M) N). Now, we are ready to define the semantics of a λ-expression. Let β(f) be a single application of β-reduction on f. If there are no "apply" nodes on which β-reductions are possible then β(f) = f. The **semantics/meaning/value** of a λ-expression e is defined as the fixed-point of the function β applied to e at the start, i.e., the value x in the sequence β(e), $\beta(\beta(e))$, $\beta(\beta(\beta(e)))$, ..., for which x = β(x). The order of applying β-reductions is not significant [2]. The end result is the same, especially if the simplification process terminates.

3 Command Line Interface (CLI) Tool

The command line tool, built using Python, allows the users to compute free variables in a lambda-expression, generate an alpha equivalent lambda expression, perform a substitution on a lambda expression, and apply beta reductions on lambda expressions. The command line tool also contains all the necessary functionality that serve as the back-end for the visualization tool described in the next section. The back-end functionality includes the following:

1. the generation of an initial expression tree for a given Lambda expression.
2. the generation of the next expression tree given the previous expression tree and function application node IDs which can be beta-reduced.

3. the generation of the next expression tree given the previous expression tree and "operator" node IDs which can be simplified.

Lambda Expressions, Grammar, Parsing. To generate the initial tree, we use a parser generator library, PLY [8], and code the following grammar rules:

```
expr : NAME
expr : ( LAMBDA NAME expr )
expr : ( expr expr )
```

The above three grammar rules completely define the syntax of the original Lambda Calculus. To make the tool more interesting, we allow arithmetic operations between numbers and thereby include the following two additional types of Lambda expressions:

```
expr : NUMBER
expr : ( OP expr expr )
```

The parser generator validates the syntax of input Lambda expressions and, if valid, produces a tree structure encoded in a Python list.

CLI Tool in Action. The following transcript from the command line illustrates free variable calculation, alpha equivalent expressions, and application of substitutions.

```
$ python3 Lambda.py

LAMBDA> fv[((lambda x x) x)];
Free variables:  {'X'}

LAMBDA> alpha[(lambda x (x x)),u];
(LAMBDA U (U U))

LAMBDA> ((lambda x x) x) [x=(lambda x (x x))];
((LAMBDA X X) (LAMBDA X (X X)))

LAMBDA> exit;
$
```

By default, the command line tool takes as input a lambda expression and produces a final beta reduct expression after repeatedly applying beta reductions until no more reductions are possible. The following is a sample run illustrating the default behavior of the tool.

```
$ python3 Lambda.py

LAMBDA> (((lambda f (lambda x (f x))) (lambda y (* y y))) 12);
```

```
INPUT:  (((LAMBDA F (LAMBDA X (F X))) (LAMBDA Y (* Y Y))) 12.0)
BETA REDUCT:  (* 12.0 12.0)

ANSWER:  144.0
```

In case the user wishes to see the intermediate steps in the beta reductions, they can invoke the tool using the `-d` option as is shown in the following transcript:

```
$ python3 Lambda.py -d
LAMBDA> (((lambda f (lambda x (f x))) (lambda y (* y y))) 12);
INPUT:  (((LAMBDA F (LAMBDA X (F X))) (LAMBDA Y (* Y Y))) 12.0)

BETA:  ((LAMBDA X ((LAMBDA Y (* Y Y)) X)) 12.0)
BETA:  ((LAMBDA Y (* Y Y)) 12.0)
BETA:  (* 12.0 12.0)

BETA REDUCT:  (* 12.0 12.0)

ANSWER:  144.0
```

Another feature of the tool is that the user can provide a startup file named `.startup-lambda` in the current directory containing names for commonly used lambda expressions as shown below.

```
true = (lambda x (lambda y x))
false = (lambda x (lambda y y))

not = (lambda p (lambda a (lambda b ((p b) a))))
and = (lambda p (lambda q ((p q) p)))
or = (lambda p (lambda q ((p p) q)))
```

The tool, when invoked, will read these names and recognize them in lambda expressions. The following is an illustration of the use of names in beta reductions:

```
$ python3 Lambda.py
LAMBDA> ((and true) false);
INPUT:  (((LAMBDA P (LAMBDA Q ((P Q) P))) (LAMBDA X (LAMBDA Y X)))
        (LAMBDA X (LAMBDA Y Y)))

BETA REDUCT:  (LAMBDA X (LAMBDA Y Y))

ANSWER:  (LAMBDA X (LAMBDA Y Y))
```

The example illustrates logical operator “and” on inputs `true` and `false` which returns a value of `false`.

4 Interactive Visualization Tool

The user interface of this interactive tool has been developed using Dash [7], a powerful framework that offers a range of library functions for creating web components with ease, while effectively managing various underlying engineering tasks. By invoking back-end functions *get_initial_tree* and *get_next_tree*, the front-end seamlessly interacts with the back-end. Within the Web interface, four key components play a crucial role.

The first component is a text box, allowing users to input Lambda expressions, accompanied by a submit button. Clicking the "submit" button invokes the back-end's *get_initial_tree*() function. Additionally, it performs necessary adjustments of the resulting tree to ensure compatibility with the front-end, then stores the processed tree within a 'Store' and a 'Canvas' component.

The second component of the front-end is the Canvas. Since the back-end represents Lambda expressions as trees, converting them into the required rooted tree format for display purposes is a straightforward process. To accomplish this, we leverage the "igraph" package [4] and employ the Reingold-Tilford algorithm [10] for tree layout. We also use Python's NetworkX package [6] for intermediate tree representation. To visualize the tree, we utilize the 'Graph' component provided by Dash. To store the current tree, we take advantage of 'Dash's' client-side, in-memory storage solution known as a "Store". However, it is important to note that the contents of the 'Store' will be lost if the browser is refreshed. Whenever another function stores a tree in this in-memory component, an intermediate function is triggered, promptly displaying the updated tree.

The third component allows users to click on and interact with the graph nodes. Within this component, there exist two types of nodes: 'apply' nodes and 'operation' nodes. The back-end incorporates separate functions, namely *get_next_tree*() and *get_next_tree_after_math*(), to handle these distinct node types. These back-end functions perform the necessary computations to generate the appropriate tree, which is subsequently sent to the front-end. As previously discussed, the front-end stores the received tree in the in-memory storage, leveraging the 'Store' component.

The fourth component, the back button, enables users to navigate back and revisit previous trees. To achieve this functionality, we maintain a list of trees that the user has encountered, utilizing the 'Store'component once again. The list of trees is organized as a stack data structure, where the top of the stack represents the current tree. When the user clicks the back button, the top of the stack is popped, triggering the new top of the stack to be sent to the 'Store'. Consequently, the new top of the stack is displayed in the 'Canvas'component, allowing users to view and interact with the previously visited tree.

System in Use. Upon submitting a Lambda expression, the user will be presented with an expression tree with 3 types of interior nodes (Lambda nodes - triangles, apply nodes - green or red circles, and arithmetic operator - squares) and 2 types of leaf nodes (variable and number nodes - rectangles). At any

step the user may "click" on a green "apply" node to perform a beta-reduction. The user may also click an arithmetic operator node to "simplify/evaluate" the expression at that node. The system performs the operation and produces a new expression tree. The user can continue with these interactions until the Lambda expression cannot be reduced any further, thereby giving the final value of the expression. At all times the "string" equivalent of the expression tree is shown at the bottom. There is a "back" button that takes the user to the previous tree to possibly try out a different "apply" operation. User "clicks" at all other areas of the expression tree are ignored. Consider the lambda expression: `(((lambda x (lambda y (+ x y))) 10) 5)`. The initial tree produced by our system is shown in part (a) of Fig. 2. A visual interactive expression tree is shown with "green" function application nodes that can be clicked by the user to perform β-reductions. In the initial tree, there is only one "green" node. Arithmetic operator nodes are not yet ready to be simplified. Upon clicking the only green node, a β-reduction is performed and the new tree is shown in part (b) of Fig. 2. Once again, we have an expression tree with just one green node. The user click on this node generates the next tree in the evaluation process and is shown in part (c) of Fig. 2. This tree has an arithmetic operator node that can be simplified. Upon clicking the arithmetic operator node, the final tree in the evaluation is shown in part (d) of Fig. 2.

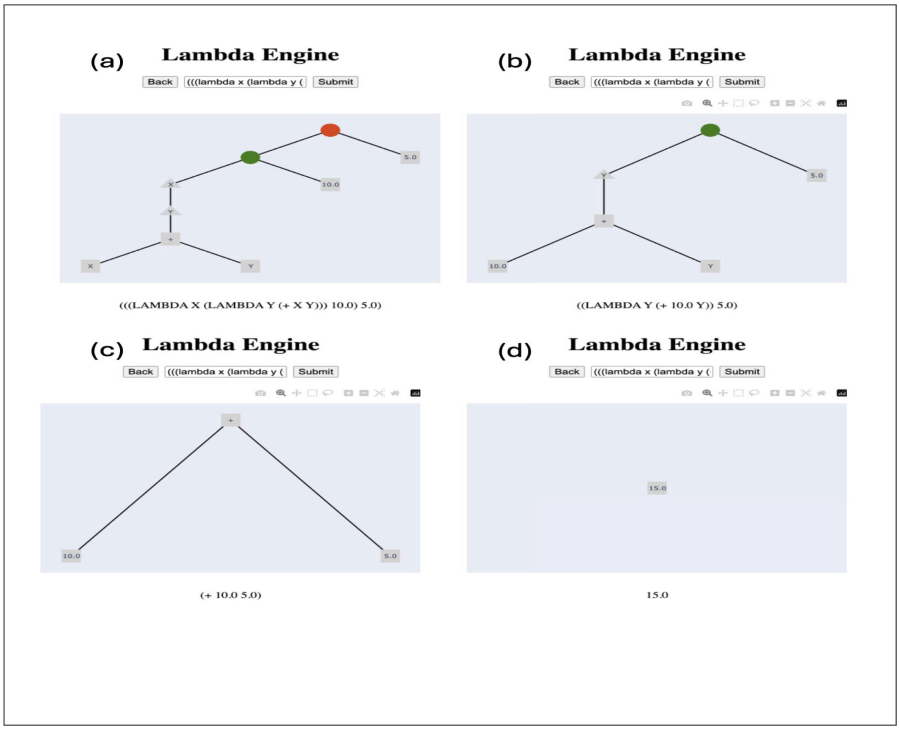

Fig. 2. Expression trees for evaluating `((lambda y (+ 10.0 y))) 5.0)`

5 Evaluation

An evaluation study was conducted in a recent offering of Programming Language Concepts, a junior/senior-level class in the undergraduate (B.S.) Computer Science curriculum. This course covers the syntax and semantics of programming languages and the various programming paradigms such as functional, logic, concurrent, and object-oriented. Lambda Calculus is introduced prior to the module on the Haskell functional programming language. Two versions of the tool were introduced:

1. the command-line version that allows the user to try out various algorithms such as determining free/bound variables, computing alpha-equivalence, applying substitutions, and applying beta-reductions
2. the dynamic interactive visualizer tool that allows the user to interactively apply beta-reductions

At the end of the module, the students were given a problem set to solve by hand and verify their results using the tools. The average grade on the homework was 80%. In addition, a survey was conducted, results of which are shown in Table 1. A total of 31 respondents out of a class of 60 completed the survey. The survey consisted of 5 questions related to the usefulness of the tool, and a free input question asking for comments/feedback on the tools. Overall, the survey indicates that the tools were appreciated by the students, with about half the students finding them "absolutely" useful and another half finding them "somewhat" useful. In the case of the tools for applying beta-reductions, about two thirds of the students found them "absolutely" useful, versus the other third finding them "somewhat" useful. In all cases, only one to three students did not find the tools useful.

In response to the free input question: *Please provide any additional feedback on the tool, including improvements*, most of the written comments were positive, e.g., "awesome tool", "wish we can see more such tools in classes", "I thought the tool overall was great!", "It was a cool tool seeing the tree form", "It was really easy to use and easy to understand!", "Loved using the visualizer especially with how its interactive", "The tools for beta reductions were good". There was one negative comment: "It's not really helpful".

Some suggestions for improvement were given: "I would provide a link to a video explaining the dynamic visual tool that goes into a very in-depth use of it", "I think a better explanation in free/bound variable in the slides would've maybe made the tool a bit better", "some documentation covering the syntax for using the interpreter would be a nice improvement".

Table 1. Survey Results

Question	Not at all	Somewhat	Absolutely
Did the use of the command line Lambda Calculus tool help improve your understanding of the Lambda Calculus concepts?	1	15	15
Did the use of the dynamic Visual Lambda Calculus tool help improve your understanding of the Lambda Calculus concepts?	1	14	16
Please indicate the usefulness of the tools for the concept of Free/Bound variables.	3	15	13
Please indicate the usefulness of the tools for the concept of Substitutions.	1	14	16
Please indicate the usefulness of the tools for the concept of Beta-reductions.	1	11	18

6 Related Work

In this section, we review several Lambda Calculus visualization tools that have been developed in the past. We compare each with our tool and present pros and cons.

Project Ultimatum's Lambda Viewer [9] provides a simple interface wherein users can submit λ-expressions and choose a particular order of evaluation such as normal order, call-by-value, etc., and the system displays a static image containing the expression tree. The system also displays the λ-expression which may be clicked to view the next λ-expression after a single β-reduction. The system seems to have similar capabilities as our tool, however, our tool has a dynamic tree visualization that the user can interact with. In contrast to Lambda Viewer, which is deployed behind a Web server (cgi-bin), our tool can easily be deployed in a Python environment.

Lambda Calculus Calculator [5] is an online tool that enables users to enter λ-expressions and submit for evaluation. The system allows the user to choose the number β-reductions to apply and see the result. The system is linear and does not include a visualization of the expression tree. Users can choose between several strategies of evaluation including innermost and outermost applications. This system is deployed on a Web server and is written in Haskell.

David Ruiz and Mateu Villaret [11] present "TILC: The Interactive Lambda-Calculus Tracer", a user graphical application that helps students learn basic concepts of untyped Lambda Calculus. The user interacts with a tree structure of the Lambda terms and the interactions are reflected in the Lambda terms.

This tool is built as a Desktop application written in Haskell and currently the download is not available at the link provided in the paper.

LambdaLab [3] is another interactive Desktop application that provides the ability to view Lambda Calculus terms graphically and manipulate them via β-reductions. The visualizations are not interactive but employ different coloring schemes to distinguish active parts of the expression tree.

Compared with the above systems, our interactive visualizer for Lambda Calculus has the following advantages:

1. Our system is light-weight and built with Python's universally available packages that can be easily installed in the present and foreseeable future, as long as Python is still a commonplace language.
2. Our system is also highly interactive, with the user under control on which β-reduction to apply, thereby allowing the user to explore different paths to the final reduced form of expressions. This allows the user to try out different evaluation strategies such as inner-most, outer-most, call-by-value, call-by-name, etc.
3. Our system is built using the Plotly graphics package, that uses WebGL or SVG to render the objects. This allows for large λ-expressions to be rendered with zoom in and out possibilities, thereby allowing users to look at different parts of the tree in detail.

7 Conclusion and Future Work

In this paper, we have presented an interactive visualization tool for Lambda Calculus. The tool provides a platform for computer science students to learn about the syntax and semantics of Lambda expressions. A classroom evaluation study was conducted and the results from the study were encouraging and we hope to make the tools available publicly.

As far as future enhancements of the tool, we envision several possibilities including showing the user inner details of β-reductions, exporting a linear trail of reductions starting from the initial expression to it's final value, providing explicit documentation on Lambda Calculus as well as the visualization tool in a Website, and provide other interfaces for the user to try out intermediate steps such as α-reductions, substitutions, and determining free and bound variables in Lambda expressions. The back-end functionalities are already built in for these enhancements and we will need only to build the user interfaces for these enhancements. The source code for this project is available at https://github.com/rajhub28/lambda-animation.

References

1. Church, A.: A set of postulates for the foundation of logic. Ann. Math. **33**(2), 346–366 (1932). http://www.jstor.org/stable/1968337
2. Church, A., Rosser, J.B.: Some properties of conversion. Trans. Am. Math. Soc. **39**(3), 472–482 (1936). https://doi.org/10.2307/1989762

3. Sainati, D., Sampson, A.: LambdaLab: an interactive λ-calculus reducer for learning. https://2018.splashcon.org/track/splash-2018-SPLASH-E?. Accessed 10 Oct 2023
4. igraph: Python-igraph, a Python interface to igraph for manipulating graph data (2023). https://python.igraph.org/en/stable/. Accessed 18 Aug 2023
5. Buchli, L.: Lambda calculus calculator (2023). https://lambdacalc.io/. Accessed 10 Oct 2023
6. NetworkX: Network Analysis in Python (2023). https://networkx.org/. Accessed 18 Aug 2023
7. Plotly: Dash: low-code framework for rapidly building data apps in Python (2023). https://dash.plotly.com/. Accessed 18 Aug 2023
8. PLY: Python Lex-Yacc (2023). https://www.dabeaz.com/ply/. Accessed 18 Aug 2023
9. Project Ultimatum: Lambda Viewer (2023). https://projectultimatum.org/cgi-bin/lambda. Accessed 10 Oct 2023
10. Reingold, E., Tilford, J.: Tidier drawings of trees. IEEE Trans. Softw. Eng. **SE-7**(2), 223–228 (1981). https://doi.org/10.1109/TSE.1981.234519
11. Ruiz, D., Villaret, M.: TILC: the interactive lambda-calculus tracer. Electron. Notes Theor. Comput. Sci. **248**, 173–183 (2009). https://doi.org/10.1016/j.entcs.2009.07.067. https://www.sciencedirect.com/science/article/pii/S1571066109002904. Proceedings of the Eighth Spanish Conference on Programming and Computer Languages (PROLE 2008)
12. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proc. Lond. Math. Soc. Second Ser. **42**, 230–265 (1936)
13. Wikipedia: Lambda Calculus (2023). https://en.wikipedia.org/wiki/Lambda_calculus. Accessed 18 Aug 2023