LambdaLab: Interactive λ-calculus for Learning

Chirag Bharadwaj
Cornell University
cb625@cornell.edu

21 May 2017

Abstract

Lambda calculus underscores some of the fundamental core of functional programming languages. Its simplicity and power often work as a useful model for semantic analysis. However, some of its abstraction often pose challenges in the classroom. Arguably some of the pedagogical methods by which lambda calculus is instructed to newly-minted functional programming students is wanting, despite the existence of suitable methods of learning. A key challenge is the **representation and processing of the calculus via a visual tool** so that students can utilize optic facilities more readily.

This proposal introduces a new method of presentation that could aid in students' understanding of the lambda calculus. In particular, existing methods of lambda calculus analysis are combined with handy visualization techniques to make the material more palatable and intuitive. The hope is that this will both engage student interest as well as provide pedagogical value in the classroom.

General Terms Learning, Visualization, Classroom

Keywords Lambda calculus, beta reduction, abstraction, intuition, TypeScript, visualization, pedagogy

1. Introduction

This project is largely motivated by the lack of a current pedagogical system for teaching the lambda calculus in the classroom beyond arcane mathematical representations. One of the primary goals is to combine current ways of analyzing the lambda calculus with unique classroom techniques for visualizing concepts. This would enable students to better understand what is really going on with the core ideas. There are many possible use cases for a system like this, but ultimately its value is most nearly academic; in particular, such a framework offers much to students who may otherwise struggle with abstract reasoning. This paper focuses on the theoretical design of such a framework and analyzes its various components.

2. Proposal and Design

The initial proposal for this project was to develop this framework over the course of a semester-long project, but due to time constraints, the project was revised to instead thoroughly discuss a theoretical implementation of such an application. This discussion was intended to survey current methods of analysis, teaching, and understanding, and determine how best to build a framework using the results. The sections below aim to piece together this information and utilize it to best service the needs of students in the classroom.

Anecdotally, one of the biggest complaints that students have when learning the lambda calculus for the first time is how to properly deal with abstraction in an intuitive and consistent way. For example, students often get tripped up by lambda expressions such as

$$(\lambda x.\lambda y.x)\ (\lambda y.y).$$

Students often make mistakes with expressions like this due to the capture-avoiding substitution semantics, which may trip up the evaluation process. For example, students will often confuse the y in the second parenthesized expression with the y in the **first** parenthesized expression when making substitutions, and end up with something like this:

$$\lambda y.\lambda y.y,$$

which is technically correct, but may be hard to interpret for first-time lambda calculus students. In fact, one of the key steps that students often forget is to rename the variables to avoid any clashes:

$$(\lambda x.\lambda y.x)\ (\lambda z.z).$$

Then, the analysis of the expression via reduction is quite simple (up to α-equivalence):

$$\lambda y.\lambda z.z.$$

This second expression is, of course, equivalent to the more confusing one presented earlier, but makes analysis a little clearer without resorting to superfluous parentheses or a discussion of right-associativity.

One of the goals of the project is to examine what exactly students find challenging about relatively sim-

ple examples like this and find alternative ways of presenting them so that students know how to proceed in a logical, deterministic fashion without having to tell them a bunch of complicated mathematical principles while still remaining fairly rigorous. A secondary goal might be to observe the mathematical capture-substitution/reduction rules but provide visual intuitions for what they do and why they work. The section below discusses (in detail) how to realize such a framework in theory.

3. LambdaLab

As discussed above, a challenging aspect of the lambda calculus for many first-time students is handling the mathematical abstraction head-on while attempting to perform analysis on relatively simple expressions. To this extent, LambdaLab, which is a proposed interactive framework for student-learning of the lambda calculus, is a viable solution. LambdaLab aims to visually bridge the gap between the confused student and the lambda calculus' various properties, all while maintaining the rigor necessary to understand the lambda calculus once the crutch of visualization is removed. The goal is to provide a means by which students can realize their understanding and clear up confusion.

However, this does come at the price of the fact that once sufficient intuition has been developed, the visual model becomes obsolete. That is, as students use the visual model more and more and are able to draw natural connections in the underlying mathematics, the visualization becomes less necessary for quick computation. Regardless, its use in developing intuition largely supercedes dealing with pure mathematics, so I posit that it is useful to create such a framework despite its diminishing returns in use over time. The methods by which such a system is achieved are manyfold, and some of them are caricatured below.

3.1. Basic Visualization

The basic visualization that LambdaLab should employ, at a public interface level, comes in the form of a text box that users can type into. Perhaps it looks something like this (the identity expression is always entered as the default expression):

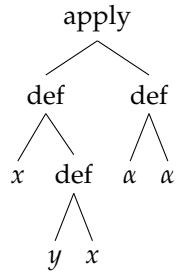
Users should be able to type closed-form lambda expressions using standard ASCII text, with perhaps the following suggestive meta-syntax:

- The \ character corresponds to a λ
- The remaining characters are one-to-one with their traditional lambda-calculus counterparts

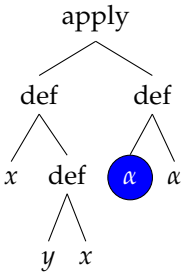
Since students often struggle with the visualization aspect of lambda expressions, one helpful method of understanding what is going on may be to produce a graphical AST of the entered expression. For example, let us say that the user types in the lambda expression from section 2:

`(λx.λy.x) (λy.y).`

Then, the following AST could be created and displayed on prompt:



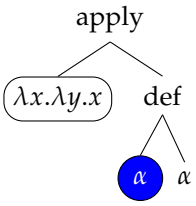
Notice how the second lambda-term had its variables automatically renamed so that capture-avoiding substitution would work properly. A notice should be given to the user in a case like this so that he or she understands the discrepancy between what is entered and what is seen. One way this could be done is to highlight the variable at define-time and notify the user. For example, it could look like this:



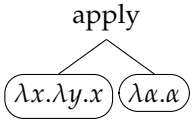
Open hovering over the circled node, the user could perhaps see a message like this:

`Your variable` x `was consistently renamed to` $\alpha^{[?]}$.

This could alert the user to be careful when performing his or her own analyses (the [?] allows the user to see what this type of message is intended to convey). Keep in mind that such a structure is just a pure visual cue. To make things a little more intuitive, having "collapsible" sub-expressions is helpful. For example, one could click nodes to collapse and expand terms as appropriate. An example of a collapse on the left child of the root node in the tree on the previous page may look as follows:

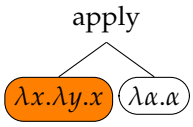


A good way to roll out this visualization may be a top-down approach, so that the entire tree is not visible at once. Rather, only the top-level sub-trees are shown upon initial generation:



Such a collapsed presentation makes it clear (in almost plain English) exactly what is going on: two lambda terms are being applied to each other. If the user then chooses to expand a sub-tree, he or she will be able to see more and more of what is going on underneath. From an implementation perspective, it can also free up rendering time by allowing only part of the tree to be shown at a time (unless more is explicitly asked for by the user).

Another subtle feature that would be incredibly useful would be synchronized highlighting. For example, let us say that the user highlights over one of the lambda terms in the collapsed tree, shown in orange:



Mapping this to the corresponding sub-term in the originally-entered lambda expression could be useful. For example, it could correspond to the following:

$$\Big(\boxed{\lambda x.\lambda y.\lambda x}\Big)\ (\lambda y.y).$$

The other suggested useful features all build carefully on top of this simple, light-weight interface.

3.2. On-demand Reduction

A useful additional proposed feature for LambdaLab is the inclusion of a choice of *reduction strategies*. A good implementation might have a switch near the input text-box where the user can choose between a call-by-value (CBV) reduction strategy and a call-by-name (CBN) reduction strategy. The hope is that once a student becomes acquainted with the intuition of how lambda expressions are constructed, he or she can play with the intricacies of eager and lazy evaluation to fully realize how different strategies can lead to surprising intermediate results.

For example, the following radio buttons can be provided next to the expression text box:

It is important distinguish reduction from visualization, however. The framework should provide visualizations on-demand, as mentioned in the previous section (just showing the top two sub-trees is sufficient). However, for reduction, it may be helpful to let the students work out the reduction processes on their own before "checking their answers". To this extent, we want to institute two different modes: (i) *answer mode* and (ii) *checker mode*. In the answer mode in CBV, LambdaLab simply lists the reduction steps from the original lambda expression to a normal form. This is illustrated in the following complicated example (moreso than the running example from Section 2):

```
(λy.(λx.x) y) ((λx.x) (λx.x)).
```

Under CBV answer mode, the reduction shown might look like this:

$$\begin{aligned}
&(\lambda y.\lambda x.y)((\lambda x.x)(\lambda x.x)) \\
\xrightarrow{\alpha}\ &(\lambda y.\lambda x.y)((\lambda \alpha_1.\alpha_1)(\lambda \alpha_2.\alpha_2)) \\
\xrightarrow[\text{CBV}]{1}\ &(\lambda y.\lambda x.y)(\lambda \alpha_2.\alpha_2) \\
\xrightarrow[\text{CBV}]{1}\ &\lambda x.(\lambda \alpha_2.\alpha_2) \\
\xrightarrow{\alpha}\ &\lambda _.\ \lambda \alpha_2.\alpha_2
\end{aligned}$$

Note that steps in which α-equivalence (not just β-reduction) is used are also shown. Just for completeness' sake, in CBN answer mode, it might show something like this instead:

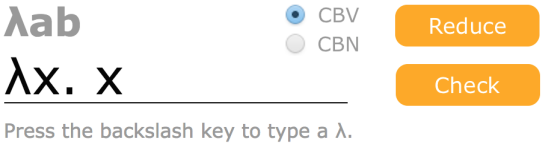
$$\begin{aligned}
&(\lambda y.\lambda x.y)((\lambda x.x)(\lambda x.x)) \\
\xrightarrow{\alpha}\ &(\lambda y.\lambda x.y)((\lambda \alpha_1.\alpha_1)(\lambda \alpha_2.\alpha_2)) \\
\xrightarrow[\text{CBN}]{1}\ &\lambda x.((\lambda \alpha_1.\alpha_1)(\lambda \alpha_2.\alpha_2)) \\
\xrightarrow{\alpha}\ &\lambda _.\ ((\lambda \alpha_1.\alpha_1)(\lambda \alpha_2.\alpha_2))
\end{aligned}$$

To show *confluence* (i.e. the Church-Rosser Theorem) under the reduction strategies, the user would have to then apply this lambda expression to another one to see that they both evaluate to $\lambda \alpha_2.\alpha_2$, i.e. the identity.

The other mode, namely checker mode, allows the user to test his or her understanding of lambda calculus. Similar to the input text box, the user will be able to type what they believe the (α-equivalent) lambda expression is after β-reduction under the specified strategy (at least one must be chosen, and CBV is the default). LambdaLab will then alert the user

whether his or her entered expression is equivalent to the expected one (up to α-equivalence). If the expression entered is correct, the user is presented with a check mark, but if the expression entered is incorrect, then the first parsed sub-expression that did not match an α-equivalent version of the correct answer will be underlined in red. However, the user will not be alerted to possible sources of error in his or her entered expression. Instead, he or she will be expected to analyze the other visual components to ascertain what could have gone wrong in the analysis.

An implementation of such an interface might look something like this:



Upon clicking "check" for checker mode, the user might be brought to a very similar interface as the original where he or she can type in the expected reduced lambda expression under the specified strategy:

These features, along with the always-displayed parse tree (AST) can help the user see clearly what a given lambda expression reduces down to using various strategies. It offers the opportunity to play around with expressions and see how different sub-expressions work in the various subtrees of a top-level expression.

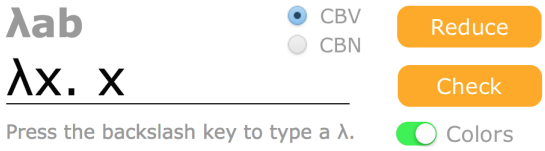
3.3. Color-coded Reduction

Of course, simply having a reduction feature may not be enough for those who are visuo-spatial learners. Another feature that is often helpful in identifying different expressions is using color-coded highlighting. To this extent, the following colors may help:

- Use black for referenced variables, e.g. x, y, z, etc.
- Use blue for function definition, e.g. $\lambda x.e$, etc.
- Use red for function application, e.g. $(e_1)(e_2)$, etc.
- Cycle through various colors when underlining sub-expressions (where sub-expressions at the same depth on the tree visualization have the same highlight stack level), e.g. (from section 3.2)

$$(\lambda y.\lambda x.y)((\lambda x.x)(\lambda x.x))$$

This type of coloring may be able to better aid students in seeing the connection between the entered expression and the tree visualization. In addition, the underlining may aid in seeing how to reduce in steps. Of course, this could get quite distracting, so opting to have a turn-off button for the colors is also helpful:



3.4. Macro Expansion and Syntactic Sugar

One of the more helpful features to add on top of pure lambda calculus is the notion of *macros*. For example, one might want to be able to *bind* expressions to names and then use refer to those bound expressions by name in some other expression. The example from section 3.2 demonstrates this:

$$(\lambda y.\lambda x.y)((\lambda x.x)(\lambda x.x)).$$

Rather than referring to the sub-expression $\lambda x.x$ or its α-equivalent versions multiple times, one can bind this to a name (the identity):

$$\mathsf{id} ::= \lambda x.x.$$

Then, it becomes easier to define the lambda expression in simpler, easy-to-understand terms:

$$(\lambda y.\lambda x.y)(\mathsf{id}\ \mathsf{id}).$$

Similarly, the CBV β-reduced expression is then

$$\lambda\_.\ \mathsf{id},$$

i.e. a *thunked* identity, and the CBN equivalent is

$$\lambda\_.\ (\mathsf{id}\ \mathsf{id}).$$

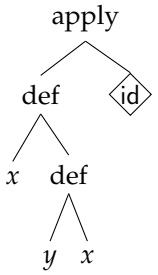
(Which has the same *normal form* as the CBV version under full β-reduction when applied to a term, by confluence.) To help make this possible for users, a useful feature for LambdaLab is to add user-side support for the binding of such macros. Each binding brings the sub-expression into the name-space (where evaluation is simply based on textual substitution prior to analysis). The syntax for defining a bound lambda-term follows exactly the same as the line with the "::=" notation above. For example, one might enter

```
id ::= \x.x
```

in the user text box. References to id are then resolved by careful CBV substitution (or simple textual substitution/full macro-expansion) in future expressions. This is done in such a way to preserve the exact meaning of the original expression while still maintaining the semantics enforced by capture-avoiding substitution. The interpreter uses *closures*, so if id were to

be redefined, then the old definition would be "forgotten" in some sense. However, if a second macro were to wrap around id prior to its redefinition, then the closure would use the old definition of id (i.e. the one available when the second macro was defined). Dynamic-patching style semantics (non-lexical closures) for macros is not available.

In the visualization, the macro is shown by default as a leaf, but it can be clicked for expansion (possibly with α-equivalent versions shown in place of the original). For example, if the expression from Section 2 were "fully expanded", it would initially show only



until the diamond is clicked on.

The following macros are pulled into the default namespace for LambdaLab and **cannot be overridden** (attempting to do so would yield an error):

- id ::= $\lambda x.x$
- omega ::= $(\lambda x.(x\ x))(\lambda x.(x\ x))$
- false ::= $\lambda x.\lambda y.y$
- true ::= $\lambda x.\lambda y.x$
- if ::= $\lambda b.\lambda t.\lambda f.(b\ t\ f)$
- and ::= $\lambda b_1.\lambda b_2.(b_1\ b_2\ b_1)$
- or ::= $\lambda b_1.\lambda b_2.(b_1\ b_1\ b_2)$
- not ::= $\lambda b.(b$ false true$)$
- pair ::= $\lambda x.\lambda y.\lambda f.(f\ x\ y)$
- fst ::= $\lambda p.(p$ true$)$
- snd ::= $\lambda p.(p$ false$)$
- nil ::= $\lambda x.$true
- isnil ::= $\lambda p.p\ (\lambda x.\lambda y.$false$)$
- S ::= $\lambda x.\lambda y.x$
- K ::= $\lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$
- I ::= id
- Y ::= $\lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$
- iota ::= $\lambda f.((f$ S$)$ K$)$

In particular, note that Church encodings of numerals are **not** actually in the default namespace.

Another topic of interest might be the notion of *syntactic sugar*. Long-form lambda expressions can be unintuitive and cumbersome to write, and while macros offer some guidance in that direction, there are other measures that one can take to simplify the notation.

For example, if multiple variables are bound by a lambda definition at once, a classic technique is to remove the separating lambdas and simply put them in scope jointly. That is, rather than using a long-form definition such as

$$\lambda x.\lambda y.\lambda z.(x\ z)(y\ z),$$

the traditional shorthand (i.e. *currying*) is

$$\lambda xyz.(x\ z)(y\ z).$$

A useful feature in the user interface would be to support this by allowing the user to bind multiple variables simultaneously in braces (with a separating space between each variable). For example, for the lambda expression above, the user could enter it as

```
\{x y z}.(x z)(y z).
```

The braces are required in case some larger expression contains the multiple bindings as a sub-expression (it is not strictly required during parsing; this just simplifies the implementation-side view a lot, though).

3.5. Pedagogical Features

Ultimately, the primary purpose of this undertaking is pedagogical, so having a few more features for usage in the classroom would ultimately prove fruitful. Among these, a particular useful one might be the ability to generate permalinks to created expressions and include a "share" button. This could enable students to be able to share expressions and constructions on discussion forums, such as Piazza. In turn, this could allow students to more clearly see what other students or instructors are talking about when asking questions.

Another useful feature surrounding the ability to share information comes in the form of exporting. Being able to semantically export the entered lambda expression could be incredibly useful for practical analysis in the form of real programming languages. To this extent, a "export to JS" button could be incredibly useful, whereupon clicking it creates and a downloads a ready-to-go equivalent JavaScript program corresponding to the lambda expression.

3.6. Implementation

A significant implementation of such a framework can be found (currently in its very initial stages) at `https://github.com/sampsyo/lambdalab`. The code version utilizes TypeScript to implement the AST and perform interpretations over it. It uses TypeScript and JavaScript (and HTML/CSS) together to visualize the actual back-end on a hosted website.

4. Theoretical Evaluation

Since LambdaLab is merely a proposal, there is currently no data on which an implementation can be evaluated. However, in theory, there are several ways in which one can evaluate the relative success and merits of an implementation of such a framework.

For example, a simple manner in which the efficacy of such a framework can be tested is by allowing students to use it in a programming languages course and surveying whether they found it useful or not at the end of the course. Presumably, the pure LambdaLab (i.e. without the revisions suggested in section 6) has just enough features to be able to cover most students' questions about ways to think about the lambda calculus. However, it could be the case that students find a basic implementation rather wanting, so surveying their experiences can be useful in knowing whether a basic version is sufficient or more features need to be added for its helpfulness in the classroom to become much more obvious.

Another way to achieve this might be to compare certain metrics across different iterations of a programming language course. If two versions of such a course are run, one with access to LambdaLab and the other without, then (assuming that the students in the experimental group do not inform the students in the control group about the existence of LambdaLab) one can measure the number of questions asked by students about lambda calculus (after controlling for variances across different batches of students) on, say, Piazza, and see whether the number is higher on average when access to LambdaLab is not provided.

A third way might be to do a similar controlled experiment, but within the *same* classroom. Objectively, this might be possible by first giving students notes about lambda calculus without teaching them anything about it. Students can be asked to read through the notes and prepare for an exam on all of the material without external guidance. Then, after the exams are given and graded, the students can be given access to LambdaLab and repeat the process. A second exam of equal difficulty can be administered, and performances across the exams can be controlled for and compared to see if performance is higher when students have access to visuo-spatial resources such as LambdaLab.

These methods give rise to statistical ways in which one can measure the importance of LambdaLab in the classroom. Further analysis can of course be conducted once real data is collected (such as averages, variances, trends, distributions, etc.).

5. Conclusions

Overall, the lambda calculus is a deceivingly simple framework that presupposes a large amount of theoretical knowledge for the purposes of full analysis. However, in attempting to uproot the current systems for understanding it, it is evident that parts of it can certainly be carved up into manageable (i.e. "bite-sized") chunks that are palatable to beginner students of functional paradigms. If done properly, an alternate presentation of the material that does not sacrifice rigor can utilize visual cues to impart what mathematical abstraction cannot for some types of learners.

In particular, the use of color, real-time interaction, and on-demand reduction are among many techniques that could enable students to more clearly see how the lambda calculus operates on a lower level. The idea is that these tools can enable beginners to gain the standard intuitions for more complex analysis without having to drill them on mathematical abstraction, all while keeping up with the mathematical rigor found in the literature. This provides a compelling alternative pedagogical device for one of the most foundational concepts in theoretical computer science.

6. Future Work

Overall, the project met the original goals, but it could still use some improvement. To this extent, many suggested directions in which this project can be extended in the future to achieve greater results are provided below. Indeed, many facets of the various trade-offs and setbacks in such a system are clearly observable from the earlier descriptions. From this newfound experience, the following modifications to the baseline framework seem like good ways in which this project can underscore more subtle aspects of the lambda calculus:

1. **Error messages.** Currently, the proposed LambdaLab framework does not describe to the user what is wrong in his or her entered reduced expression in checker mode; instead, it merely highlights/underscores the first parsed expression that differs from the expected result (up to α-equivalence). A useful direction to take this in is to provide actual hints as to what might have gone wrong rather than just basic parser errors, as are shown currently.

   Such an extension would require looking into various common misinterpretations and hand-learning patterns of mistakes that students often make. These could then be encoded into the interpreter and suggestions could be given to the user based on which mistake is most likely to have occurred. Some common mistakes, anecdotally-speaking, that often come in the classroom include:

   (i) Forgetting to rename captured variables and getting a completely wrong answer

   (ii) Misinterpreting where implicit parentheses go and thus using the wrong associativity

   (iii) Using the wrong or mixing reduction strategies to get an answer that is close

2. **Reduction strategies.** The proposed LambdaLab only supports call-by-value (CBV) and call-by-name (CBN) semantics for reduction of lambda-terms to normal forms. An interesting way to extend the project might be to support other (possibly more arcane) deterministic reduction strategies. For example, the following additional ones could be useful if students/users want to see even more different styles of evaluation (and see more confluence!):

   - *Call-by-need.* Also known as *modified lazy evaluation*. Call-by-need semantics is a modified version of CBN in that it also **does not** reduce terms inside a function definition (unlike CBV). However, unlike CBN, call-by-need semantics **caches** the reduced function arguments so that it does not need to be evaluated every time in the body of a function. If each function argument is used at least two times, this offers a speed-up when compared to pure CBN semantics.
   - *Applicative-order.* In applicative-order semantics, CBV is mostly employed, but the arguments to a function are evaluated (i.e. reduced) in a left-to-right post-order traversal of all reducible expressions (i.e. redexes not already in normal form).
   - *Normal-order.* In normal-order semantics, CBN is mostly employed, but the outermost expressions are always evaluated first. However, function applications are evaluated before function arguments, so unlike CBN, the body of an unapplied function is still evaluated.

3. **Other visualizations.** Students may find that the current visualizations are insufficient for their understanding. Some alternate ways of looking at things may be helpful. For example, Stoy diagrams and de Bruijn notation are other classic techniques for simplifying lambda-calculus interpretations. The *Stoy diagrams* help clarify the *scope* of bound variables in a lambda expression. For example, in the lambda expression

   $$\lambda x.(\lambda y.(\lambda x.(x\ y))x)x,$$

   it may be helpful it see where each bound variable's definition is. This is accomplished via the following style of diagramming:

   It is possible to show (not done here) that two lambda expressions are equivalent (in the sense of α-equivalence) if they have the same Stoy diagram.

   Another notation that is often useful is due to de Bruijn. In particular, *de Bruijn indices* serve as replacements of when a variable was bound. Bound variables in a lambda expression are replaced with natural numbers (including 0). Each natural number corresponds to the λ symbol that binds it. Specifically, the bound variable is replaced with some natural number k if the the λ that binds it has the kth smallest scope (zero-indexed) out of all possible scopes that bind the same name. Thus, for the same lambda term above, the de Bruijn indices are

   $$\lambda x.(\lambda y.(\lambda x.(0\ 1))1)0.$$

   Then, again (proof omitted), two lambda expressions are α-equivalent if the have the same indices.

   A nice addition to the project might be to have checkboxes for displaying the Stoy diagram and/or de Bruijn indices for an entered lambda expression in the basic visualization, if students find that these formulations are easier to understand.

4. **Context visualization.** Another helpful mechanism for visualization might be to include some sort of "context pot". That is, it may be helpful to see what he or she has bound or is currently in scope in an on-demand basis as they write lambda expressions. This can be done by having some sort of visual manifestation of the current dictionary of bound variables and their values. This "mixing pot" would hold all of the bound variables currently in scope for wherever the cursor is in the user text box, which can enable the user to correct some mistakes if he or she misinterprets certain implicit parentheses or other semantic idiosyncrasies in lambda calculus.

5. **Recursion.** Currently, recursion is supported in LambdaLab by virtue of the Y-combinator, which is pulled into the default namespace, as mentioned in section 3.4. However, it may be useful to allow for recursion via syntactic sugar. Evidently, this is only possible for macros, i.e. bound lambda terms. An example of its usefulness is shown below; rather than writing the following recursive program as

   $$\begin{aligned}\mathsf{P} ::= &(\lambda g.(\lambda p.(\lambda xyz.(x\ y)(x\ p)(p\ z)))(g\ g))\\ &(\lambda g.(\lambda p.(\lambda xyz.(x\ y)(x\ p)(p\ z)))(g\ g)),\end{aligned}$$

   it is cleaner to simply rewrite it as follows:

   $$\mathsf{P} ::= \lambda xyz.(x\ y)(x\ \mathsf{P})(\mathsf{P}\ z).$$

   Supporting this kind of syntactic sugar for recursive purposes can be useful (in that it helps remove the need for the Y-combinator except when explicitly calculating the fix-point of a lambda-term). We can do this as follows. Let us suppose that F is some recursive bound lambda term written in the syntactic sugar form, i.e. the body of F calls F itself. Then, we can bring this to a definition that uses the Y-combinator as follows:

- Define some **new** macro G, which is based on F
- G takes in an extra argument f before taking in all of F's arguments and replicating the body of F
- However, G replaces each occurrence of F in its body with just f
- Then, we redefine F as F ::= Y G

This re-ordering can allow the interpreter to easily allow for recursive lambda expressions to be bound by the user in a readable way.

6. **Church numerals.** As mentioned in section 3.4, Church numerals/encodings are **not** supported by the proposed LambdaLab. However, an (almost-expected) extension might be to bring Church encodings to the default namespace. This would involve a little bit of tricky evaluation, as users will be able to directly write numbers in the text box. However, not all of the natural numbers will be known ahead of time, so the interpreter must have some way of generating (and caching/memoizing) them at run-time so that the correct corresponding lambda expressions can be substituted. Perhaps the numeral $\overline{0}$ is hard-coded (or, even more fancily, there is *interning* for the first, say, 128 numeral encodings, since they are likely to be used the most often).

   Along with the Church encodings, the following macros are probably a good idea to pull into the default namespace as well:

   - iszero ::= $\lambda n.n\ (\lambda x.\mathsf{false})\ \mathsf{true}$
   - succ ::= $\lambda nfx.f\ (n\ f\ x)$
   - pred ::= $\lambda nfx.n(\lambda g.\lambda h.h\ (g\ f))(\lambda u.x)(\lambda u.u)$
   - sub ::= $\lambda m.\lambda n.(n\ \mathsf{pred}\ m)$
   - plus ::= $\lambda m.\lambda n.(m\ \mathsf{succ}\ n)$
   - mult ::= $\lambda m.\lambda n.(m\ (\mathsf{plus}\ n)\ \overline{0})$
   - pow ::= $\lambda b.\lambda e.(e\ b)$

   Supporting syntactic sugar for some of these functions (e.g. with standard infix operators such as `'+'`, `'-'`, etc.) should be possible with just the parser (i.e. a pre-interpretation translation can be done at parse-time to convert these into the expected constructs).

7. **Debugger.** Having a debugger similar to `gdb` for LambdaLab could prove useful as well, if students' lambda expressions are particularly complex. Specifically, it would be helpful if the framework stepped through the tree representation of a program via an in-order traversal when in *debug mode* (which can be added to answer mode and checker mode, as described in section 3.1).

   The system can display reduced expressions (under the specified reduction strategy) up until that point in the traversal on the side, so that a user can see whether this matches his or her expectations. In fact, this can be combined with the reduction-checker mode described in section 3.2. If a user wants to check what **part** of a lambda expression reduces to (in context), this can be done if both debug and check modes are set up to work concurrently. Of course, this would require an additional extension to the check mode logic to ensure that the behavior is different when the user is simultaneously in debug mode.

8. **Importing programs.** One last incredibly useful extension is the addition of the ability to import pre-written programs to the system. This can allow users to write longer programs and save them prior to uploading them so that the framework can interpret them. The actual visualization and user action post-upload will remain the same, but now programs can span several lines (new lines and whitespace are ignored). The syntax is the same as what is expected in the current version of LambdaLab, so this mostly just provides the user with a way of neatly writing the lambda calculus program in his or her own preferred style. All uploaded files should have, say, the extension `.lambda`.

7. Acknowledgements

I would like to thank my advisor, Adrian Sampson, for the opportunity to work on such an open-ended project as well as for providing key insights when I wasn't sure how to proceed with the project. His contributions were very valuable, and his endless patience and thought-provoking sessions made for a great experience. I would also like to thank Nate Foster and Michael Clarkson for fostering my interest in programming languages and its theoretical foundations in the glory days back in 2015. Lastly, I'd like to thank Andrew Myers for getting me interested in computer science in the first place. Without his interesting style of instruction, I'd probably still be choosing a major.

References

[1] H.P. Barendregt. *The Lambda Calculus, its Syntax, and Semantics*. North-Holland, 2nd edition, 1984.

[2] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[3] Dexter Kozen and Nate Foster. *Lectures in the Theory of Programming Languages*. `https://cs.cornell.edu/courses/cs6110/2016sp/lectures`. Lectures 2-3 (λ-calculus). 2001.

[4] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.