Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

A visual language for Lambda Calculus

Rónán Carrigan
[redacted]

April 28, 2020

A Dissertation submitted in partial fulfilment
of the requirements for the degree of
MCS (Computer Science)

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: [redacted] Date: 28/04/2020

Abstract

The Lambda Calculus is a broad, complex area of mathematical and computational theory. It is prevalent in society today through common functional programming languages, but is still the source of confusion for many people studying these languages. This confusion is founded upon two distinct aspects of the Lambda Calculus, the structure of the language and the computation of the language, known as beta reduction.

Visualisation is a commonly used technique to help people understand a topic more easily. Several attempts to visualise Lambda Calculus expressions and animate beta reduction have been made before. Some of these have been proposals for designs and some have been working implementations but each has issues that were not addressed by their creator.

This work proposes a new design for untyped Lambda Calculus visualisation and beta reduction animation. Each of the existing designs discussed are evaluated and contribute elements to the design. The field of data visualisation overlaps with this work significantly and so several resources were used in the design to benefit from the extensive research available in creating effective visualisations. An implementation of the design is provided that can be used online.

The system proposed acts as a foundational step in the area, leaving ample work for future research. As is typical with applications, user trials are the next step in future work so that the effectiveness of the design can be evaluated in a controlled environment.

Acknowledgements

I would like to thank my supervisor during the writing of this work, Professor G. Strong, for his essential guidance through each step of my research.

Contents

List of Figures

1 Introduction

1.1 Background and Motivation

Functional programming is a large and complex topic, with academic research spanning decades available for new readers to dive into. Due to the mathematical foundations of the subject, much of this research is not aimed at the general person studying computer science and related fields or a software developer branching out from their knowledge of more traditional programming methods. We propose that a visual application exploring the Lambda Calculus would be useful in learning the workings behind it, and so put forward a new system that could be used for such a purpose.

Functional programming has been around since the early days of higher-level programming languages. The Lambda Calculus is the model of computation behind functional programming languages and is equivalent to the well known Turing Machine[1]. Haskell Curry stated, referring to Lambda Calculus substitution, *'most formulations of the rule for substitution (. . .), which were published, even by the ablest logicians, before 1940, were demonstrably incorrect'*[2]. There is a large amount of subtlety behind this computational model.

Functional programming is gaining in popularity, as seen across popular languages adapting features and functional languages appearing in StackOverflow's most popular technology[3]. Despite this, intuitions behind these languages can be difficult to grasp and teaching resources are commonly disjointed. It can be seen in many of these resources that the authors attempt to build a visual intuition of the Lambda Calculus to build understanding. For example, Jung[4] creates a simple tree structure to represent expressions in Lambda Calculus. Trees are a natural way to represent these expressions and are often what is used but these have several issues with them that will be addressed later in this paper.

Visual programming languages have long been used to assist in the perception of

computation. Boshernitsan and Downes[5] present an extensive look into the history and use of visual programming languages. One popular example of such a language is the Scratch programming language[6]. Scratch is often used in education as an introduction to basic programming and helps to build intuition of concepts such as control flow and data mutation.

We hypothesize that a tool to visualise Lambda Calculus expressions and to animate beta reduction would be useful in comprehending the mechanics behind the language. The goals of such a tool differ from an application like Scratch where the graphical aspects are in the building of the program to be computed, which could be described as a graphical editor. The focus of such a tool instead would be on the visualisation of both Lambda Calculus computation (i.e. beta reduction) and its representation along the way. In the textual version, the semantics of reduction are impossible to display between steps, leaving newcomers to build their own intuitions around them. As an example, $(\lambda a.\,(\lambda a\ b\ .\ a\ b\ a)\,(\lambda a.a))\ \ (\lambda a.a)$, reduced using Application Order reduction, reduces to $(\lambda a\ b\ .(\lambda a.a)\ b\,(\lambda a.a))\ \ (\lambda a.a)$. Even relatively simple reductions such as this can confuse viewers as there is a significant amount of work being performed out sight. A visual tool could bridge the gaps between reduction steps, allowing users to stronly grasp the concepts at play.

To create such a tool however, a system to display Lambda Calculus expressions must be derived. There have been several attempts to create a visual representation of the Lambda Calculus in the past, each with varying degrees of success along with advantages and disadvantages. Keenan[7] outlined the specification that inspired this paper but had only created an informal version and left the full details up to further work.

1.2 Dissertation

Keenan's specifications are not, nor are meant to be, suitable for general use, but rather are intended for simple demonstration by hand. He states that for future work in this area *'the graphical notation should be formally described and the correspondences between it and the well-understood textual lambda calculus should be elucidated if it is to be widely used'.* This paper sets out to answer two questions as to its primary goal. The first being how can a set of rules be formalised to allow any arbitrary lambda expression to be drawn?

The next step after formalising the rules for drawing expressions is creating a system in which these expressions can be reduced, in a series of reduction steps.

Keenan discusses a method of tying these steps together with animations. Once again the system he puts forth is simply a theoretical one, for proof of concept purposes. Several problems arise once these methods are applied to complex expressions. The second question that is addressed is how can these reduction steps be animated to allow for arbitrary expressions?

The main use of a system as put forward in Keenan's paper would be for educational purposes. If extended sufficiently, as Keenan states, an explanation could be provided of normal order reduction versus unsafe reduction orders, head normal form, weak head normal form and the different types of reduction (Alpha, Beta and Eta). The system could be used to supplement these explanations to provide visual intuition to these mechanisms. As a secondary goal, this work attempts to provide an implementation of such a system to form the base of an educational tool, although evaluation of the efficacy of the implementation will be left for future work.

2 Background Information

2.1 Lambda Calculus

This section will serve as an introduction to the untyped Lambda Calculus, with further reading being necessary for greater understanding. Suitable resources for basic understanding include the tutorial by Rojas[8] or more in-depth paper from Sestoft[9]. Barendregt's book[10] is commonly used as a reference for a complete definition.

Structure

The untyped Lambda Calculus structure is that of a tree of nested expressions. Each expression is one of three types:

- Variables, denoted by an identifier in the form of a string of alphabetical characters.
- Abstractions, where a variable identifier is bound over another expression, denoted by a λ, followed by the variable name, then a ‘.‘ and finally the inner expression.
- Application of two expressions,

This can be represented with a BNF grammar:

$$<\textit{expression}> ::= <\textit{ident}> \mid \lambda<\textit{ident}>.<\textit{expression}> \mid <\textit{expression}><\textit{expression}>$$

Reduction

Computation in Lambda Calculus is done through reductions of expressions. There are three types of reduction

- α: Renaming of variables. Alpha reduction is required so that two expressions that have bound variables with the same name do not clash.
- β: Substituting a variable bound by abstraction by applying it to another expression.
- η: When two expressions always produce the same value, then they are equivalent and can be swapped with each other.

Beta reduction is the most important form of reduction as it is how values are calculated. A reducible expression, which will be further referred to as the commonly used name 'redex', is an abstraction applied to any other expression. To reduce the expression all instances of the bound variable within the abstraction on the left are replaced by the right expression. For example, $(\lambda a.a\ a)Y$ would be reduced to $Y\ Y$

The point of beta reduction is to normalise expressions to some point. Normalisation can be in four forms:

- Normal Form (NF): No beta reduction is possible.
- Weak Normal Form (WNF): No beta reduction is possible above any abstractions.
- Head Normal Form (HNF): No beta reduction possible on head position.
- Weak Head Normal Form (WHNF): Head position is not a redex.

Head position is the leftmost, outermost expression.

Reduction Strategies

There have been many beta reduction strategies derived, each with different approaches and goals[9][11]. Some goals of strategies include simplicity of reduction, minimising duplicated work or guaranteeing reduction to normal form if possible. Normal Order and Applicative Order reduction will be discussed here to provide a sample of varying approaches.

Normal order reduction works by reducing the leftmost, outermost expression redex at each step. In practice, this means that arguments to abstractions are substituted before reduction. Normal Order reduction will eventually reduce to a normal form if one exists. In opposition to this, Applicative Order reduction reduces the leftmost innermost redex first, meaning that arguments to abstractions are evaluated before

being substituted for bound variables. By default, Applicative Order is not a normalising strategy and so does not guarantee to find an expression's normal form. Some modifications can be made to change this such as in Sestoft's Hybrid Applicative Order[9].

Encoding and Computing Values

As is typical in mathematics, the Lambda Calculus uses only simple constructs which can be used to build all other needed data types and functions. There is no notion of numeric or data primitives built into the syntax. Church describes a method of encoding values using recursive higher-order functions, later known as Church Encoding, and that any computation that can be modelled in a Turing machine can also be modelled as a Church Encoding[12]. There are other methods for encoding[13] but they will not be discussed for simplicity.

Examples of Church encoded numbers are as follows:

$$0 := (\lambda s\ z.z)$$

$$1 := (\lambda s\ z.sz)$$

$$2 := (\lambda s\ z.s(sz))$$

A number n can be thought of as a function that will apply its first argument n times to its second argument.

Data constructors can be made by collecting elements and deciding how to apply functions over it:

$$TUPLE := (\lambda\ a\ b\ f.f\ a\ b)$$

Arithmetic can be encoded using successor functions as a basis:

$$SUCC := n.s.z.s(nsz)$$

$$PLUS := \lambda m\ n\ s\ z.m\ s\ (n\ s\ z)$$

2.2 Related Work

The related work that we will discuss will be all related to the visualisation of Lambda Calculus and animation of beta reduction. Each work works toward a similar goal of a tool to be used for education as opposed to more complex uses by more knowledgeable users. The depth of relevant work varies, with some like Keenan, creating simple hand-drawn styled depictions of lambda expressions, while others have produced working code for animations of beta reduction. In this section we shall outline the main aspects of the designs under discussion, however, for full details on each, the corresponding text should be read.

To Dissect a Mockingbird -David Keenan[7]

Keenan describes his system as 'an informal and entertaining introduction by means of an animated graphical notation'[7] to the Lambda Calculus. Keenan uses the bird metaphor originally developed by Smullyan[14] for describing lambda expressions, where the songs of the bird are their values. Abstractions are a box with a left semicircle being its 'ear' where it listens for or receives an input and a semicircle on the right which represents its throat where it sings or outputs a value. Inputs and outputs are shown by lines drawn between these 'ears' and 'throats'. Variables are represented by a single circle with the inputs that bind them connected to them by straight lines. The application of expressions is drawn by connecting the output of one expression to the input of another, again with straight lines.

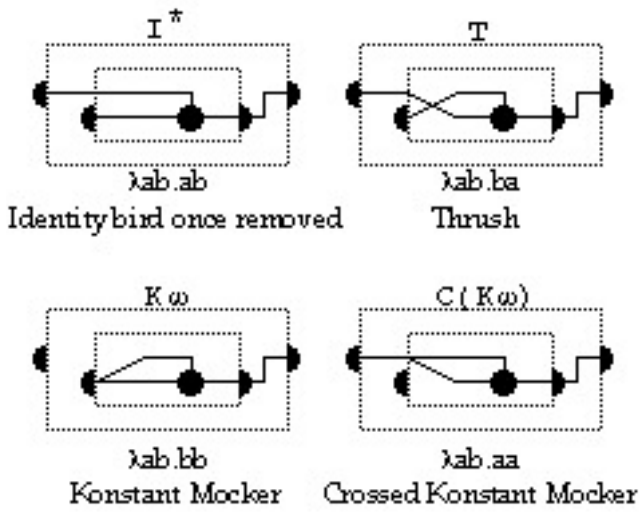

Figure 2.1: Examples of Keenan's drawings along with their expressions

Keenan also addresses a way in which to animate these diagrams and presents these

in the form of a film reel style set of slides to illustrate the movement of expressions along application and binding lines. These are, however, as he states, only an informal specification and not designed to be used in a working model but rather as a starting point.

The base expressions of Lambda Calculus are given, which can construct any other expression and basic arithmetic is also discussed. There are no extensions made to the language, in the form of constructions such as true, false or tuples, which are present in other works.

Programming with Visual Expressions -Wayne Citrin et al.[15]

Citrin et al. have similar goals to Keenan in that they propose that the system could be 'used in teaching the concepts of lambda calculus as a replacement for or augmentation to the teaching of traditional textual rewrite rules'[15]. The group make no mention of Keenan's work so it is uncertain if they were aware of his design but they do take a different approach to Keenan, by using nested circles to draw expressions. They provide drawings of their design and name their visual language VEX but provide no implementation. Variables are of the form of empty circles and abstractions are circles containing others, with their bound parameter internally tangent connected by lines to their bindings. Application of expressions is indicated by arrows from left to right, with the precedence of application being denoted by numbers.

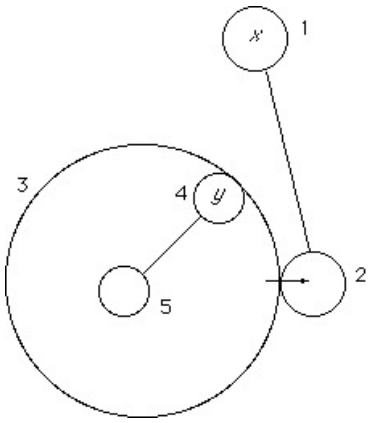

Figure 2.2: Examples of VEX drawing for expression $(\lambda y.y)x$

Rudimentary, predefined animations are supplied to demonstrate how a system could implement these rules. Although they are not of high quality, they do show how the VEX language allows for smooth transitions between reduction steps.

Citrin et al. describe how the VEX language allows for extra constructs such as booleans and numbers. They also add in the ability to represent a let expression, which is sometimes added as a fourth type of expression in Lambda Calculus but is not necessary as the other three are. Let expressions are of the form *let X = Y in Z* in which all occurrences of X in Z are replaced by Y when being reduced. Once any use of X is reduced, the other occurrences are also replaced since they will reduce to the same value. Variable binding like this is used to reduce duplicated work when performing beta reduction.

Visual Lambda Calculus -Viktor Massalõgin[16]

The system created by Massalõgin, called Visual Lambda, is the first discussed that has been implemented with working animations. It also uses circles or rings to represent variables and abstractions. Each bound variable is a different colour, with rings of the same colour being the same variable. Application is done through adjacent rings, with precedence determined by the expressions overlapping where the uppermost expression is consumed first.

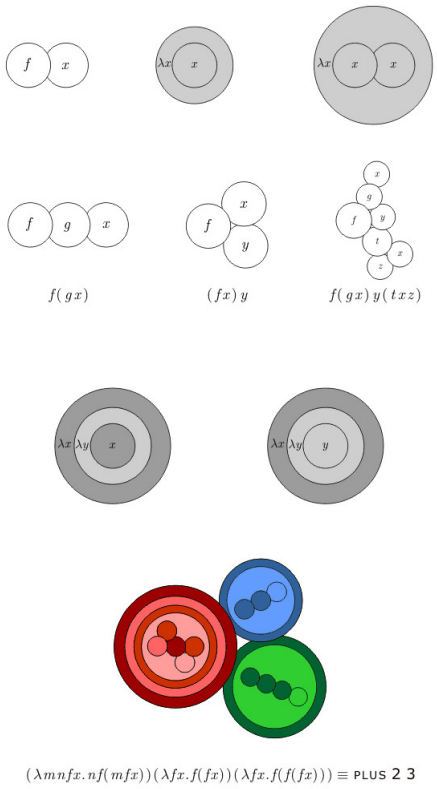

Figure 2.3: Examples of Visual Lambda expressions

Source: https://bitbucket.org/bntr/visual-lambda/wiki/Home

The program to run the animations can be run locally using Python 2. Full descriptions of the animation details can read in the paper, but some of the advantages and disadvantages will be discussed in the design chapter.

Visual Lambda also allows for the use of let expressions. These are done by colouring multiple rings the same until they are dereferenced. The use of other constructs was not implemented in the project but Massalõgin did suggest labelling rings as synonyms for different values and functions.

Lambda Animator -Mike Thyer[17]

The Lambda Animator by Thyer is another working implementation of Lambda Calculus visualisation, however, it takes a considerably different approach to the previously discussed ones. Thyer uses a tree graph system to display the expressions. This type of graph is what is traditionally used to represent a Lambda Calculus expression visually since expressions are inherently trees by nature. All expressions are nodes on the graph and are distinguished by the text within them. The rules for these labels are complicated and not necessary to explain fully here. Further details can be found on the website for Lambda Animator.

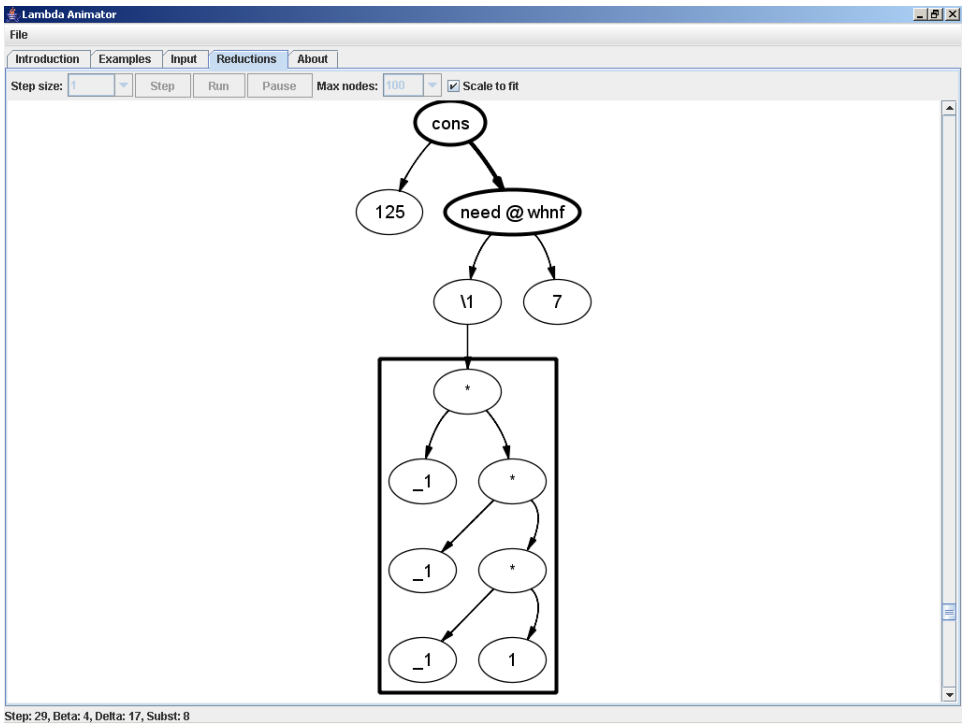

Figure 2.4: Examples of Visual Lambda expressions

Source: https://bitbucket.org/bntr/visual-lambda/wiki/Home

Since labels are used for nodes, the system allows for constructs such as numbers to appear as built-ins for the language. Unfortunately due to the age of the project, the implementation was unable to be tested and thus no evaluation of the visualisation could be made beyond the images provided. The traditional tree-based approach to this visualisation is certainly familiar to those in the field and for good reason. The simple layout and explicit relationships between expressions are easy to parse for single static diagrams but its suitability for generalised use and animation is not so certain.

3 Analysis & Requirements

Each of the existing designs brings forward distinct approaches to the problem of visualising static expressions and Massalõgin provides a working implementation of animated beta reduction. The goal of the work in this paper is to use the existing design features and synthesize several improvements over each. While the final design will attempt to achieve this, the requirements put forward to take into account the fact that user trials will be necessary to effectively evaluate the system. Therefore they generally focus on being configurable and extensible over well-defined bounds.

User interface and data visualisation guides will be referenced often in the following chapter to facilitate the discussion and critique of existing designs, as well as justify choices made determining requirements. The works of Ward et al.[18] and Smith & Mosier[19] are in use here as they are used as standard resources in education and offer a vast range of guidance in design and data visualisation. The paper of Yi et al.[20] was chosen as it offers an in depth exploration of the possible categories of interaction that are prevelant in data visualisation, which is highly relevant to the design of the system.

3.1 Overlapping Elements

Position is the most important of the visual variables and thus the final design should be heavily influenced by the following requirements. Ward et al. state that 'The best positioning scheme maps each graphic to unique positions, such that all the graphics can be seen with no overlaps'[18, Section 4.3.1]. An issue can be seen in Keenan's and Citrin's design as the lines used for application and bounding inherently overlap over all of the other expressions. Thyer handles this requirement through the use of an expanding tree and therefore having no overlapping elements at all. He uses text to highlight the binding of variables which has the disadvantage

of forcing the user to rely on text which can be difficult to interpret, as Smith and Mosier[19, Section 2.4/1] express 'People cannot readily assimilate detailed textual or tabular data' Massalõgin avoids using binding lines but does overlap expressions to show application order.

Application order and variable binding seem to be the root of the issues with overlapping elements. The design will need to handle these issues in a way that avoids the pitfalls seen in others. The only exception to this rule is when elements are in motion where overlapping is preferable to fading out and into position to allow tracking of movement.

3.2 Spatial Limits

Another aspect of positioning is accounting for the space in which data is displayed. Spatial limits are an essential factor in the positioning of data. With Keenan's, Massalõgin's and Thyer's designs, the issue of dealing with expanding expressions is not addressed. Once the expression gains significant complexity, each of these becomes unwieldy and fail to summarise the expression in any sort of way. Citrin provides a solution to this problem. In his design, he proposes the use of primitives such as constants and operators. This not only reduces the complexity of large expressions but also allows the user to understand the semantics of an expression without needing to know the internals.

Another solution to this problem would be to allow the user to focus on specified subsets of the root expression. This allows them to analyse areas of interest without having to filter out the noise of the rest of the system.

The design should utilise at least one of these solutions but ideally allow for both. It should also allow for scaling of the graph to the user's preference so that they can view both large and small expressions on the screen.

3.3 Distinct Shapes

'Mark' refers to the shapes used in the visualisation.[18] Ward et al. explain 'it is important to consider how well one mark can be differentiated from other marks'[18, Section 4.3.2]. As Smith and Mosier[19, Section 2.4/12] state, the use of symbols within a system should be consistent. Symbols should not have multiple meanings to not confuse the user.

Massalõgin and Citrin use circles to represent both variables and abstractions. Although the two expressions are distinguishable due to other symbols within abstractions, this can still obfuscate the meaning behind the diagrams. Thyer's system suffers the most as he uses nodes in the tree to represent variables, abstractions and applications, relying on text to distinguish each. Keenan manages to keep separate shapes for variables and abstractions but does use connecting lines to represent both application and variable binding.

The design should use distinct shapes for each type of expression but also for any other elements present in the visualisation.

3.4 Clear Application Order

Describing the application order effectively is essential as it fundamentally changes the meaning of expressions. Showing the precedence of operations must be done but also easily introduces a lot of noise into the system. In the textual lambda calculus, this is done with parentheses, which quickly balloon expressions into unwieldy masses of parentheses.

This issue is tackled differently by each system. Citrin uses textual notation to describe precedence. This has the disadvantage of forcing the user to rely on text which can be difficult to interpret, as Smith and Mosier[19, Section 2.4/1] express 'People cannot readily assimilate detailed textual or tabular data'

Massalõgin uses the height of circles to show precedence. Once more than 4 or 5 variables are present in an expression it becomes awkward to parse which expressions are applied where and is not ideal for quick reading.

Keenan and Thyer appear to be best in this case as the use of lines allow for clear reading of precedence even at a glance, although the lines in Keenan's have issues as discussed previously

The design should allow the user to easily parse the application order while also handling the spatial issues with the existing approaches discussed earlier.

3.5 Colour Support

The ability to support colour allows users to easily distinguish between different points of interest. It is important to note the difference between support and

reliance. Massalõgin uses colour in his system but it uses very similar colours for adjacent expressions, which becomes difficult to read when tightly grouped. The three other systems make no mention of colour in their systems.

Ward et al. state 'When working with categorical or interval data with very low cardinality, it is generally acceptable to manually select colours for individual data values'[18, Section 4.3.5]. The challenge here is that the number of data values is theoretically limitless. Therefore, if colours are to be used, the selection of them will need to be automated. The issue of unique colour generation is complex and in the case of inability to guarantee uniqueness, there should be additional mechanics in place to supplement the information supplied by colour. One possible way to do this would be with the use of textures to replace colours.

3.6 Labelling

Labelling allows the user to distinguish between expressions and also for additional language primitives as demonstrated by Thyer. The issue with labelling is that it can become excessive if used heavily and the advantage of distinguishing expressions is nullified due to the number of unique labels the user needs to track[19, Section 2.4/1][18, Section 9.3.1]. Thyer and Citrin both rely on text to different lengths, but without it, their system cannot function.

Massalõgin and Keenan both avoid the use of text in their visualisations, but it can be seen how each could add it to their design. Massalõgin demonstrates this in his future work discussion, showing the potential use of labels for primitives. This is facilitated by the fact that parents encompass their children and thus can act as a black box in which inputs enter but the inner workings are unknown. Similarly, Keenan's system would allow for the same kind of labelling.

Another potential use case of labelling would be to track variable names so that users can see the link with the textual and visual representations.

The design should allow for labelling of variable names. In the case of primitives being implemented as discussed in spatial limits, they should be labelled so the reader can easily understand their semantics. It would also be useful if the user could create new primitives in some way with custom names.

3.7 Smooth Motion

Motion can occur in any of the other seven visual variables, changing their appearance over time. The final design could utilise motion in more than just changing of position. Smooth motions are vital to the ability of the user to follow along with the reduction steps, as Ward et al. while discussing changes in data where the user needs to maintain context, state 'it is best to provide smooth transitions between the initial and final visualizations'[18, Section 12.7].

Massalõgin's system, as the only one we could evaluate in action, worked well in this regard with a clear focus on providing smooth transitions. Unfortunately, as stated in the Future Work section, some areas that did not quite reach the desired goals in this regard. One other issue found that wasn't pointed out in the work was that the way application works means that bubbles shrink into nothing from one place and expand into place from another. The sudden change in positioning is jarring for the user, especially when multiple variables are replaced at once.

Keenan provides basic animation frames to illustrate how it would be done and these do show how application could be performed very smoothly, without any sudden changes to expression positioning.

Since Thyer's animations could not be reproduced and no samples were provided it is not possible to comment on the effectiveness of the system. It is possible to imagine how one could perform these or Citrin's animations, similar to Keenan, by following the lines drawn for application and binding.

The system should use smooth motion when animating any changes.

3.8 Interaction

User interaction is a fundamental element of many modern learning environments, from classrooms of young children to online courses for software development. For example, Scratch[6] allows users to drag and drop blocks of instructions for the program to execute. Unfortunately, the potential of interaction is an area not discussed by either of the static systems of Keenan and Citrin. Thyer also does not discuss any sort of interaction from users in his writings and again this can't be evaluated by running the program.

Interaction can come in numerous forms, as discussed by Ward et al.[18, Chapter

11] Massalõgin does introduce several interactive elements to his design. Users can select any section of the displayed expression and the textual form of the selected section is printed to the console. This allows users to quickly parse the relationship between the textual and graphical versions. There are also controls along the side of the display which allows for editing of the displayed expression. The efficacy of these controls in creating greater understanding is not certain as this separates the graphical expressions from the textual ones.

The design should allow for several types of interaction so that the user has a greater level of comfort with the system. As much interaction as possible should be optional as overloading the user with duties will also lead to a poor experience.

4 Design

The design of the system was inspired by the previously discussed attempts at Lambda Calculus animation, taking aspects from each while also focusing on new features and improvements. The language overall can be classified as a space-filling method of describing hierarchical structures, designed specifically to accommodate not just visualisation of static expressions but also the animation of reduction.

4.1 Static Elements

Relationships between expressions are expressed through their position on the graph as opposed to connecting lines or some other element. These elements add noise to the system where they can be represented through implicit means while also making the goal of reduction animation much more complicated to achieve as there are more variations to consider. This is why a system using a traditional recursive tree would be very difficult to implement. Therefore, the only explicit elements are those that are lambda calculus expressions or primitives.

Figure 4.1: Y Combinator $(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$

The basis of the static layout is to simulate the textual lambda calculus to allow users to easily convert between the two forms. The Y Combinator is a fixed-point combinator which allows for the use of recursion in the Lambda Calculus. An example of the Y Combinator can be seen in Figure 4.1. The semantics of these diagrams are explained in the following sections.

Base Elements

Variables are the simplest element in their design. They simply consist of a single colour circle. The colour of the variable is determined by if it is bound by an abstraction or not. Free variables are uniformly coloured black, while bound variables are coloured to match their abstractions so that each instance of a variable is filled with the same colour.

Figure 4.2: Bound Variable

Figure 4.3: Free Variable

Abstractions contain an input 'socket' and output 'plug' closely resembling Keenan's 'ear' and 'throat' design. The input is coloured to show the colour of the variable that is bound by the abstraction. As well as the input and output, the abstraction uses an encompassing border around its inner expression.

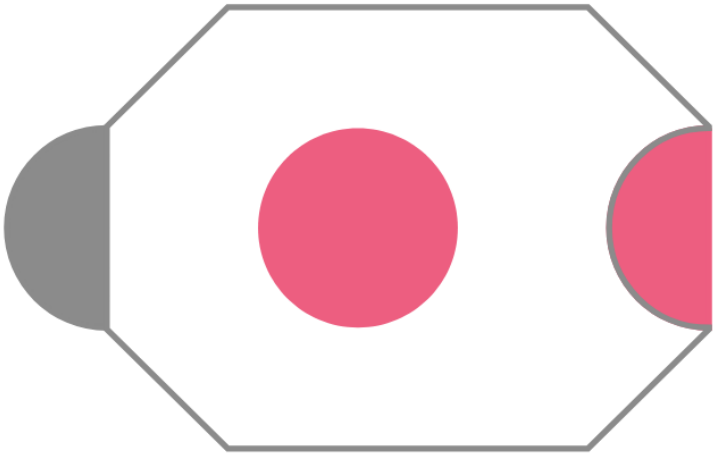

Figure 4.4: Abstraction equivalent to $(\lambda\, a.\, a)$

Application expressions use a similar design to abstractions, but they lack an input slot to demonstrate to the user that they receive no data and can only be reduced. Explicitly showing applications as a separate type of expression demonstrates to the user that they are an expression rather than some other type of primitive to the language. In the existing designs discussed they are displayed more as a

relationship between expressions which can suggest a distinction between applications and abstractions/variables. The edges of applications and abstractions are designed to emulate the use of parentheses in the textual version.

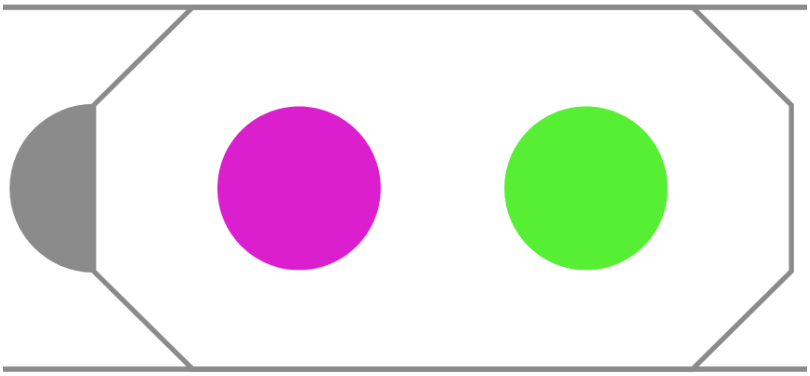

Figure 4.5: Application, within another expression

Primitives

Primitives mimic the shape of abstractions though the styling is different. They each have an output and input but the input is empty since the variable bound by the outer abstraction is not visible. The body of primitives is the same colour as the borders of other expressions to not add to the number of colours that must be distinguished by the user.

Each primitive is assigned a name, a string of upper-case letters, that is written in the body. These names are the only use of labelling relied upon for identifying information of an expression. Other uses occur as discussed later but are only supplementary.

Figure 4.6: Primitive for the addition function

Direction

The direction of flow in the system closely follows that of the textual version. Arguments to abstractions are on the right-hand side of them and thus application is left-associative. A significant difference to the textual version is that the bound variable is declared on the right-hand side of the expression as opposed to the left. The reasoning behind this is to more closely reflect the nature of how reduction works and thus more easily allow the animation of inputting variables which is shown later.

The direction of flow was also a problem in the original draft designs of the system where abstractions and applications had straight vertical lines as opposed to angled ones on their edges, which can be seen in Figure 4.7. The issue with this is that it is difficult to parse whether these lines are representative of an opening or closing of an expression despite the presence of input and output elements, especially concerning applications.

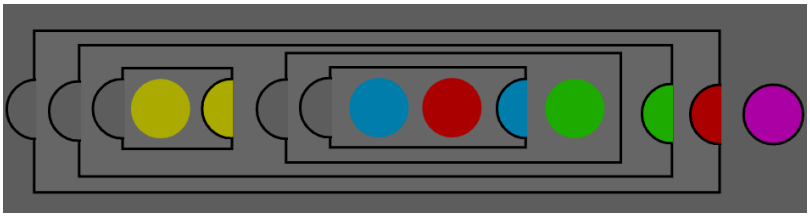

Figure 4.7: Draft of the system during the initial design

Reducing Noise

In the initial design of the system abstractions and applications had padding between their upper and lower edges and their children, visible in Figure 4.7. This had the advantage of showing the scope of each easily but once expressions became deeper than four or five levels, there were a large number of distinct elements on the graph and the whole system was difficult to parse.

To solve this, expression heights were all made to be equal constants. This means that there are overlapping elements which is not ideal for visualisation but it is minimal and constant across expression sizes while eliminating a larger issue that grows with expressions.

Within the textual Lambda Calculus, there are also two common methods to reduce the number of elements present in expressions. One of these is to remove unnecessary parentheses around applications where the order of application is not

changed through their presence. For example, $(\lambda x.\ \lambda y.\ (x\ y)\ x)$ could be written as $(\lambda x.\ \lambda y.\ x\ y\ x)$. The same optimisation is applied to this design, where applications are only drawn where they have an effect

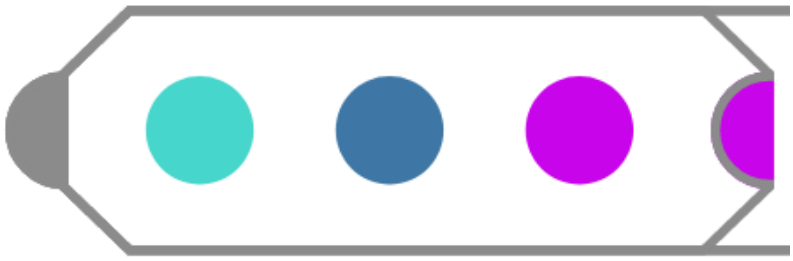

Figure 4.8: The application of the first two expressions is not shown.

The second reduction of elements in the text is the joining of nested abstractions so multiple variables appear to be bound by one and thus are multi-argument functions. For example, $(\lambda x.(\ \lambda y.\ x\ y))$ could be written as $(\lambda x\ y.\ x\ y)$. To achieve a similar effect nested abstractions are joined at their outputs, which considerably reduces the size of multi-argument functions while also demonstrating to the user the relationship between parent and child abstractions.

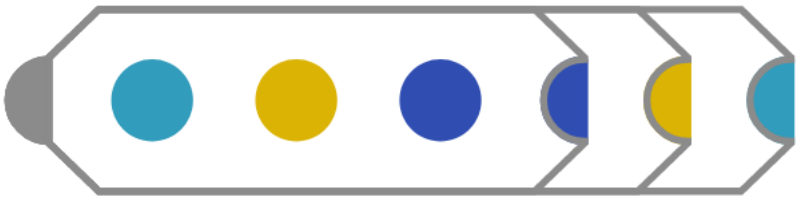

Figure 4.9: Three abstractions joined together.

This effect is also used on nested applications and works in the same manner. Interactivity is used to handle issues with difficulty parsing the scope of expressions and is discussed later.

4.2 Animated Reduction

Beta reduction animation is broken down into several steps, each with individual and distinct actions. These steps are designed to draw the focus of the user to a single point at a time to not overload them. Although in implementation they are individual stages, the whole reduction occurs in a single smooth process. All changes in expression location are done using smooth transitions, meaning there is no popping in and out of expressions so that the changes can be easily tracked. For

demonstration, we shall walk through a single reduction of the expression $(\lambda a.\ (\lambda a\ b\ .\ a\ b\ a)\ (\lambda a.a))\ \ (\lambda a.a)$ using Application Order reduction, which was also discussed in the introduction of this paper. The animation steps of this reduction can be seen in Figure 4.11, which are labelled and occur is numerical order. These labels will referred to in the description.

Figure 4.10: Expression to be reduced: $(\lambda a.\ (\lambda a\ b\ .\ a\ b\ a)\ (\lambda a.a))\ \ (\lambda a.a)$

To draw the attention of the user initially to show where the reduction is occurring, the consumed argument is highlighted by changing the border colour to black (**Step 1**). The output 'plug' of the consumed expression is then slotted into the input 'socket' (**Step 2**). This motion is designed to build an intuition of the inputs and outputs are related.

After the argument is consumed, duplicates are made to match its appearance. The original argument is kept in place to show the user that the created expressions are not the same as the one being consumed, although they still have the same colour to not confuse the user by mixing up colours. These duplicates are then raised over the argument, showing the copying of the argument to the user (**Step 3**). Each duplicate moves to position itself directly over the variable that is being replaced (**Step 4**). This is where the user first sees the separate instances of new expressions created rather than a single one.

Bound variables that are being replaced are faded out of the graph to allow the duplicates to fill their space (**Step 5**). Each duplicate expression now descends into their position, with the encompassing expressions expanding their size to fit them if necessary (**Step 6**). The binding abstraction and consumed expression are now faded away from the graph (**Step 7**). Finally, to complete the reduction, the remaining expressions on the graph are adjusted to move into place if necessary. The duplicated expressions change to different colours to show that they are now unrelated (**Step 8**).

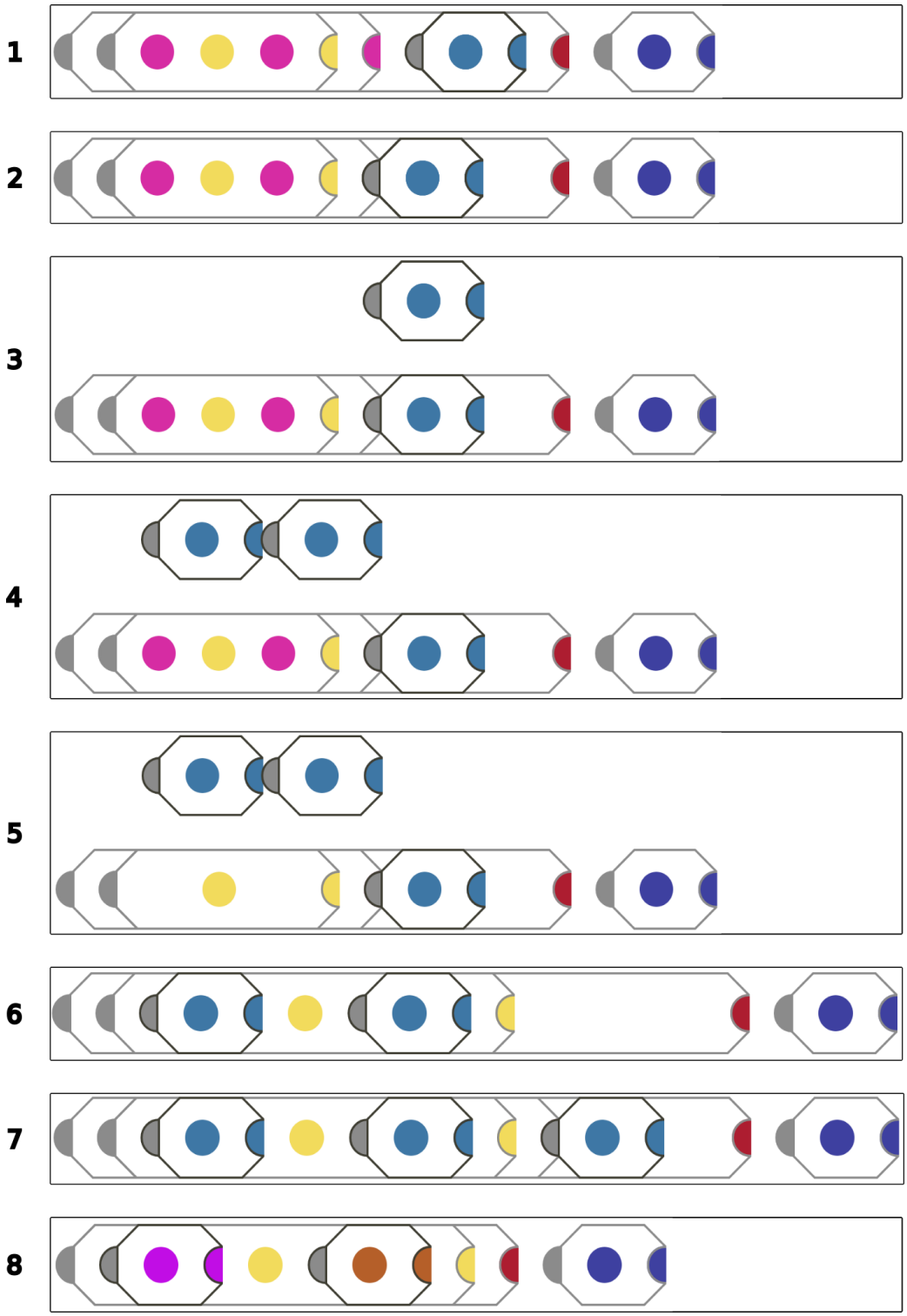

Figure 4.11: Stages of reduction.

4.3 Controls

Interaction is a fundamental part of the design as a way to mitigate issues that occurred in existing works or that would arise without their presence in this system. The first instance of interaction in the system is within the control panel.

Expression

The textual representation of the current expression of the graph is displayed in a small text-box to the user as a way to maintain the relationship between the two representations in their mind. The length of the text box is limited to avoid distracting the user in the case of large expressions.

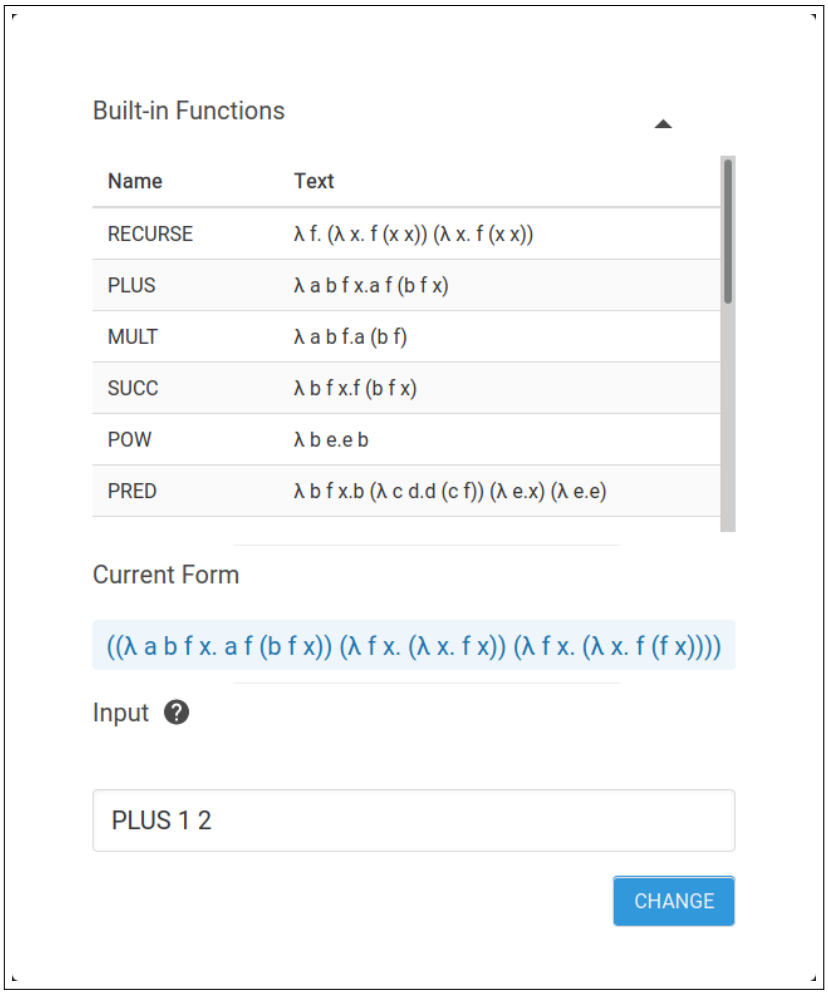

Figure 4.12: Modal for expression control.

Clicking on the box will show a modal containing three sections. The first contains a dropdown of descriptions for the built-in functions of the language which can be

used in expressions by the user. Each one consists of an upper-case name and the expression that they represent. Below is the current textual representation of the expression being shown on the graph. This is a larger box to view it than in the control panel and also allows scrolling if the expression is too large. At the bottom of the modal is the input area for creating new terms. The rules for the format of the input are standard with examples provided by the built-in functions. For convenience, the λ character can be replaced by a backslash character '\'. Built-ins are simply inserted using the upper-case names and will appear on the graph as primitives.

Arbitrary numbers can be inputted as digits and will be evaluated to the corresponding Church Encoding also being shown as a primitive.

Appearance

The correct configuration of the visualisation's appearance is a subjective matter that is likely better determined by the user to their needs. To enable customisation, the control panel contains sliders for the size of the expressions on the graph and the speed of beta reduction animation. Since the number of expressions on screen can vary so drastically, changing the size of expressions is essential for fitting the currently viewed graph to the screen. Changing of the animation speed is allowed so that novice users can pace the animation to their liking to not be overwhelmed, but also to allow more experienced users to evaluate larger expressions without needing to follow every single reduction step.

Primitive

As discussed, built-in functions are shown as primitives but users can also create custom primitives. By selecting an expression, which is discussed later, the user can assign it a name. The expression is then changed to fit the new primitive and the name displayed in the centre.

Using primitives like this allows for the 'abstract/elaborate'[20] category of interaction and means users can control the level of detail in the graph at any one time.

Reduction

The reduction control is placed at the bottom of the control panel. There are three buttons, the flow control, which allow users to perform a single reduction, reduce the expression until the normal form is reached or to step back and undo the previous reduction.

Undoing reductions occur in the same manner as reduction except in reverse. This ability enables the user to replay reductions easily while maintaining the same context that would be lost by reloading the page or entering the original term again.

Underneath the flow controls is the selected method of reduction. The currently selected method is displayed and the user can select a different one from a dropdown that appears. Currently, there are options for Normal Order and Applicative Order reduction to demonstrate the differences in reduction methods.

4.4 Graph Interactivity

The second instance of interaction in the system is with the graph and its expressions themselves.

Expression Selection

Expressions can be selected by the user through hovering over or clicking the expression. Selected expressions have their stroke changed to black, which contrasts the selected expression.

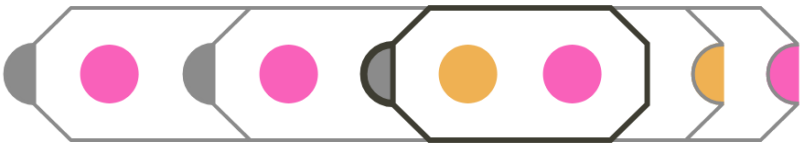

Figure 4.13: Selected application expression.

As Yi et al.[20] describe, selection allows users to track items of interest, especially during changes in view, in this case, expressions during reduction. If an expression is part of the consumed expression, then the corresponding duplicate expressions are also highlighted to allow the user to follow where the duplicates are placed.

Determining the scope of expressions can be an issue when expressions become complex since the user must match the corresponding opening and closing elements of many expressions. Another way selection can be used by the user to track expressions is in immediately determining the size of expressions without having to perform any mental work.

Expression selection also fits into the 'connect' category of interactions[20]. When an abstraction expression is selected, all variables bound by that expression are also highlighted to show the connection between the expressions.

Tooltip

A tooltip appears underneath an expression when it is selected by the user, containing information concerning the expression. These tooltips display the type of expression selected along with its textual representation. For variable expressions, the name of the variable is shown and for abstractionsa, the name of the bound variable is shown.

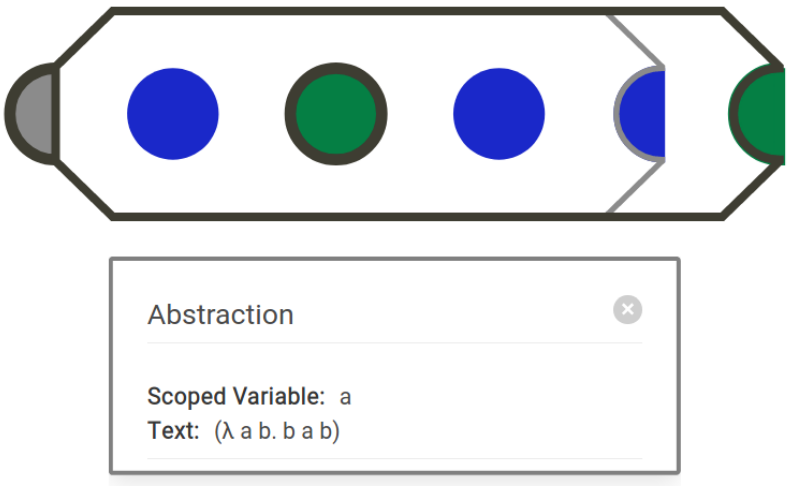

Figure 4.14: Tooltip for an abstraction expression.

The tooltip falls under the 'encode' category of visualisation, displaying the same underlying data differently, specifically as text. Since one of the goals of the system is to help users gain an understanding of the textual representation of Lambda Calculus, it is important that they can relate the visualised expressions to the text at any time.

If the expression selected is not at the top of the tree, the tooltip allows the user to focus on that expression only, which removes any expressions not below the selected

expression in the tree. This acts as a 'filter' type of interaction. Allowing focus on specific nodes means that complex expressions can be parsed in steps without overwhelming the user, or if certain sections are of significance then the user can do more than selecting the expression. Once the user is finished observing the focused

If primitives are present, the user must be able to reveal the underlying expressions. When a primitive is selected, the tooltip allows the user to de-construct the primitive so that the expression underneath fills its place. This forms the elaboration portion of the 'abstract/elaborate' interaction that is provided by the use of primitives.

Panning

Although the controls allow the user to change the scale of the graph if expressions are too large to fit the screen, sometimes the user needs to view large expressions in detail. The solution to this issue is the ability to pan the graph around the screen. Panning is a form of the 'explore' type of interactions, allowing users to navigate the graph as they wish.

5 Implementation

5.1 Base Technology

This section will highlight the main third party technologies used in the creation of the system. The advantages of the libraries used will be discussed in detail later on. Recording their use here is to provide the reader with a context for the rest of the chapter.

React JS

The application is implemented using the React JavaScript framework[21]. A web-based solution was chosen because web browsers provided a powerful graphics environment out of the box without needing to worry about cross-platform support and the site can be hosted elsewhere giving the greatest possible access to the application for users. React allows for a declarative programming style to easily create interactive UIs within the browser. Since the design requires smooth animation, high performance is an essential characteristic of whatever technology used. The library uses the idea of components for reusing elements and is heavily optimised around this concept. A render cycle is used where components are called with the required parameters each frame. React optimises this process by only re-rendering components whose parameters changed from the last cycle.

The ecosystem around the framework is one of the largest and most mature in interface design. Many libraries exist in the areas of design, animation and data visualisation. These properties made React the clear choice in the creation of the system.

In addition to React, TypeScript is used as opposed to regular JavaScript. TypeScript provides much greater developer tooling to issues and provide guarantees to the code.

Motion Control

There are many mature animation libraries for JavaScript and React, each with advantages. Several were examined for use in this system but the one that was chosen was Framer Motion[22]. Framer Motion is a physics-based library that approaches animation differently to traditional methods. Where traditionally the duration of animation is used to control speed and feel, physics-based libraries control the duration of animation by changing several properties of animated elements. This allows for easy tweaking of the animation settings for all elements at once, meaning that the user can use these controls with ease.

Another advantage of physics-based libraries is the use of 'spring physics' which means that animated elements are not just taken from point to point but also emulate real-world characteristics of moving objects with weight and resistance. This is used in this application to provide more 'natural feeling' transitions between states.

Many of the physics-based libraries for React share similar APIs to Framer Motion, but Framer Motion has many including stability, consistent updates due to corporate backing as well as functions that allow for tracking of animations as they execute which is essential for controlling the sequence of complex animations.

State Management

Management of the application state is an important factor. Contained in the store is the state of the expression tree, the reduction currently in action, appearance and animation settings, the behaviour definition of graph nodes in response to user interaction, etc. To facilitate the management of this complex state, the popular Redux[23] library. This is a general JavaScript library that comes with mature React bindings.

Redux was chosen as it is accompanied with many useful tools that can be used both in development and usage. For development, Redux comes with support for browser debugging where the developer can track every interaction with the state and monitor the changes in between each one. There are also many convenience functions which remove much of the boilerplate code involved with state management seen in other methods such as React hooks.

Global storage of state is used to allow any component to subscribe to changes in state and handle whatever has occurred. This is an important design pattern used

in React and enables any subscribers to easily integrate with the state without needing to change the implementation behind it.

Interface Elements

To create attractive elements of the user interface, such as the controls, two libraries were used. Bulma[24] is a CSS framework that uses CSS classes to style components in a consistent manner. This was chosen because the use of CSS classes is simple but also provides attractive elements. For more advanced components, requiring greater control, Google's Material UI[25] library is used. These components also include JavaScript functionality built-in, allowing for animations and other functionality.

5.2 Architecture

React enforces a strict top-down application structure. Information cannot flow up through the structure of components. For an application like this one where certain data is required throughout different sections, a mechanism for sharing and updating the state consistently must be in place. As discussed Redux is used for this purpose. The application works in a classic event sourcing manner, where a global state is held and listeners subscribe to the changes. Events can be emitted from components in response to local changes either through user interaction or its internal workings.

Redux contains the visual and tree state which is passed down through to the cached selectors which receive part of the state and pass it to a function to calculate a value. The result is then cached and if the same state is passed to the selector, it will return the cached result. Selectors allow the use of stored values globally without needing to pollute the state unnecessarily.

Both the results of selectors and application state are passed down to the Graph component which encompasses the user controls and elements for drawing. Every time the user interacts with the application or the state changes, the graph is rerendered with the updated state.

The controls allow the user to set the expression being shown, speed of reduction and size of the nodes in the graph as well as create primitives. The mode for

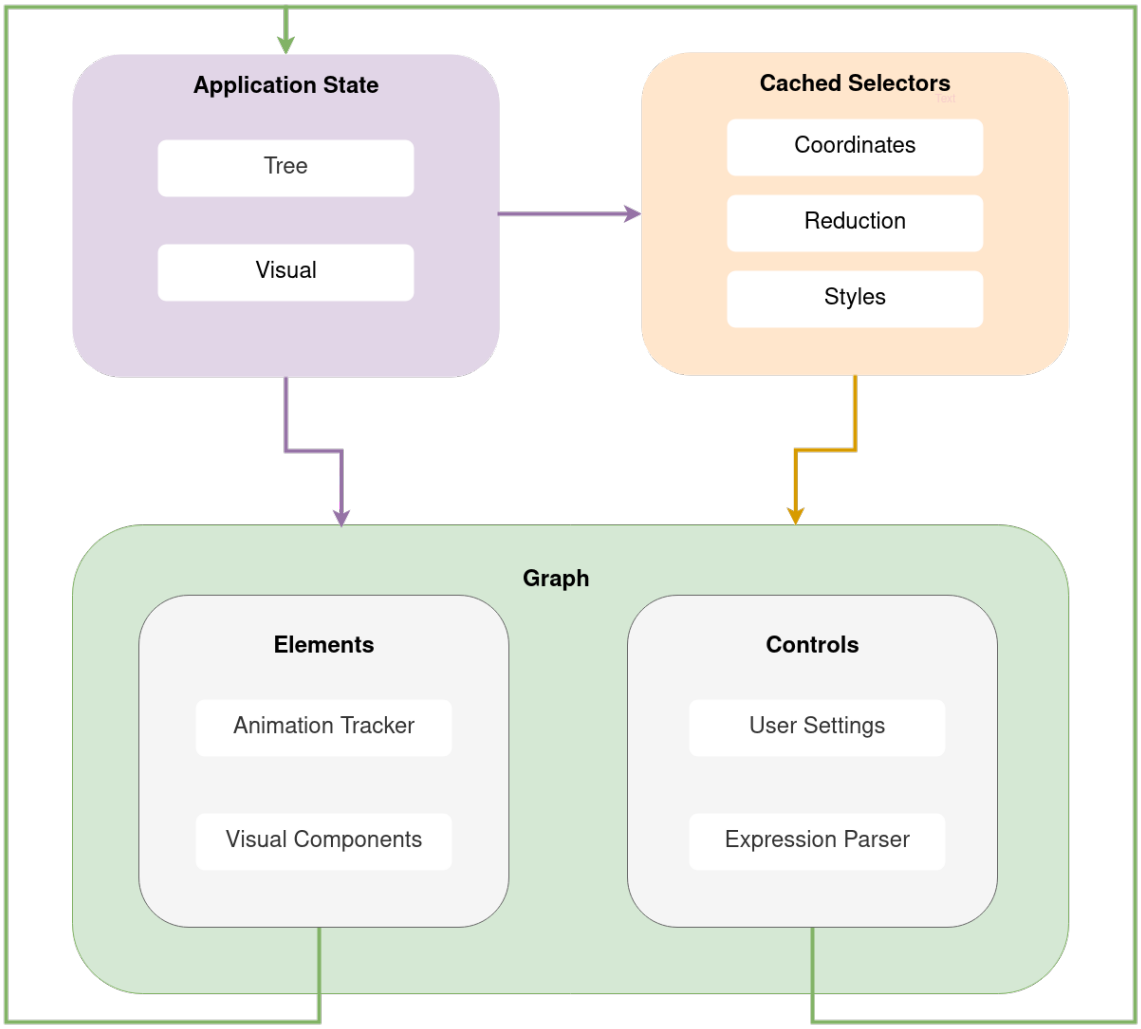

Figure 5.1: A high-level diagram of the system architecture

animation is also controlled here, where the user can select to reduce until normal form, reduce a single step, reverse a single step or stop reduction.

The elements of the graph plot the nodes of the tree using the coordinates and styles provided while also tracking the state of animations in each. Every time animations stop, according to the mode of reduction mode, the graph component decides how to proceed and emits an event to the state if necessary.

5.3 Lambda Calculus Encoding

This section will discuss the mapping of the Lambda Calculus to the TypeScript language for use later on. The work here is not providing anything novel to the area but acts as a foundation for the following sections. In the following explanations,

the use of the word node refers to individual expressions of the Lambda Calculus and the word tree is used to refer to the overall structure of the expressions together.

Node Types

The three types of nodes (Variable, Abstraction, Application) are represented through the use of interfaces. Each one has a set of functions implemented for convenience in reduction and graphing.

In the usual textual representation of Lambda Calculus, there are many problems involved with maintaining references to variables correctly as nodes are reduced and duplicated. To solve this, the De Bruijn index is used[26] internally. Variables each have an index stored to signal the abstraction by which they are bound. Unbound variables simply have an undefined index. The name that was assigned to the variable after parsing is also stored to the label for the user, but these are not used in reduction and conflicts are not checked for.

Abstractions have recorded the name of the variable they bind and then the ID of the node inside of it. Applications similarly only have the IDs of the nodes in their left and right slots.

Tree State

A sum type is used which can be any of the node types. This is used for interacting with nodes where the type does not matter. Node IDs are represented by unique strings that are randomly generated. The state of the tree is represented by a record of the root node ID, the collection of nodes in the form of a JavaScript object, commonly known as a hash map or dictionary, where entries are mapped by their ID. There is also a record of the reduction currently being performed on the tree which will be discussed later on.

The tree is not represented by a recursive structure as is usually used for Lambda Calculus. There are many situations throughout the application where the individual nodes are referred to where it is more efficient and convenient to access the node through a key lookup as opposed to traversing a tree. The root node ID is recorded for use in traversing the tree while graphing and reducing, where convenience functions that simulate tree traversal are used.

Parsing the Textual Lambda Calculus

Parsing of the Lambda Calculus is a well defined and solved problem. The parsing implementation follows standard rules of the Lambda Calculus. Variables are defined by lower case strings, whereas upper case names refer to built-in expressions. Abstractions can be started using the λ and displayed as such when rendering text but for user convenience, the backslash '\' character can be used in its place. White space is ignored except when distinguishing variable names.

The parsing library Parsimmon[27] was used to implement the parser. Parsimmon is a parser combinator library which works by using higher-order functions to combine small, simple parsers to construct larger, complex ones. As the Lambda Calculus is made up of a small number of elements combined, combinator parsing is well suited to solving the problem. The only significant hurdle in the design of the parser was eliminating the possibility of left-recursion. An expression can be evaluated as an application, abstraction or variable. If the parser decides to attempt to parse an application which starts with an expression, it can become trapped in a recursive loop and thus fail. To solve this problem, applications are only parsed within parentheses, breaking left recursion. The expression parser will then parse as many expressions as it can, creating a list of expressions. If more than one is found it will reduce across the list from the left, combining the accumulated expression and the current one under an application until there is a single expression with an application as the root. If only one expression is parsed then no application is created and the parsed expression is returned as the result.

5.4 Static Graph

The static graph was the first visual part of the system created. It relies on the previously discussed tree state and the elements discussed here.

Visual State

The visual state contains records of several items important to the display. The expression inputted by the user is kept for parsing. If a node is selected by the user or highlighted by a reduction then the ID is stored so that other elements of the UI can react.

Since the design allows users to alter the appearance of the graph in dimensions and positioning, these changes are accessible to all components to adjust accordingly. The settings of node dimensions and the position of the tree are stored with the visual state.

Node Coordinates

The calculation of coordinates is performed anytime there is a change in the dimensions or position settings or the tree state. They are not stored in state, instead, they are calculated on the fly when needed using selectors.

It begins with the calculation of the width and height of each node. To do this, the tree is traversed until a bottom node is reached, where the dimensions are calculated based on the configurable settings, therefore remaining always responsive to these changes Once the bottom-most child is calculated then the result is passed back to the parent node where the child size is accounted for. This repeats up and across the tree until every node has its dimensions calculated.

Where the dimensions of nodes depend on their children, the location of nodes depends on their parents. The location of the root node is set by the setting in the visual state and then the starting position for the child is calculated which is passed down for the child to use. The repeats across the tree until all nodes have a corresponding location and dimension, stored in a coordinate object. By using recursively calculated values, the coordinate system is trivial to change should any elements be added, omitted or changed in future work.

Node Appearance

The styling for each node is implemented using selectors similarly to coordinates. Each node type requires a different styling object. Applications have a constant appearance whereas variables and abstractions require individual colouring. An exterior colour generating library is used to create colours based off of node IDs. The use of IDs to seed generation brings two advantages. The first is that variables can use their binding abstraction's ID to create their style without needing to create the abstraction style first. The second is that using a constant seed means that colours can be generated with selectors and always produce the same value thus avoiding the need to store them in the application state.

5.5 Reduction Animation

The second major stage of implementation was the animation of reduction. Lambda Calculus reduction is a vast and ever-expanding area of research. This work is not meant to address any novel or interesting methods for reduction, instead opting to only include basic, well-known methods. How these methods are implemented will not be discussed. The implementation of reduction methods in this application is designed to allow further work to integrate with a simple interface.

Reduction State

To animate the reduction of nodes, the overall desired motion is broken into several stages. Each stage handles a single aspect of the animation. When a reduction is started, the first stage is stored as part of the tree state along with several other pieces of data. The IDs of all nodes involved in the reduction are stored so that all areas of the application using the reduction can access them without needing to calculate them separately.

A large factor in the design of the reduction system is that the Redux store must remain pure in its changes so that changes can be recorded properly. This meant that the generation of random node IDs has to be implemented outside of the store. When a reduction is started, the IDs for all new nodes created are passed in as part of the reduction information. Due to this, the calculation of reduction results is performed completely separately to the handling of the changes in the state. This is part of the reason why the addition of new reduction methods is so easy.

Coordinate Offsetting

Within the coordinates selector method, there is logic to handle the different stages of reduction. These are calculated using the bounds of the graph and configurable variables meaning that, similarly to the static graph, any changes in the appearance will be adapted to automatically. Each stage has its own set of coordinate offsets for certain nodes and some stages relocate nodes to entirely new areas of the graph. By offsetting nodes during the dimension and coordinate calculations, all nodes calculated after the offset one adjust accordingly without additional logic. This avoids the need for adjusting any nodes that are not involved in the reduction process.

Style Overriding

Styling is also overridden during the reduction of the tree. Nodes that are of interest in the reduction are highlighted to the user using overrides in the selector method. To enable the maintaining of duplicated node colours until the reduction is over, the selector method generates a mapping of node IDs to copy from. Nodes that copy another will then appear identical for the duration of their presence in the copy mapping.

Handling Changes

The Framer Motion library handles the changes of both styles and coordinates in a simple, declarative style. Values are supplied to a library method every render and the returned object is used to draw the corresponding node. When the values supplied to the library are changed, the returned object will gradually change at a rate determined by configurable values in the visual state. All moving elements are tracked using event emission. Once they all come to a rest the next reduction stage is triggered.

The rate at which changes occur is altered depending on what the current state of the graph is. When the graph is being moved around then the changes occur quickly to not appear sluggish while they are slowed down during reduction so the user can follow along.

Reduction Reversal

The reversal of reductions is where the choice of technologies brought the most advantage. Redux has an external library called redux-undo which allows for changes to be reversed on all or parts of the state. Using this functionality along with the declarative style of React, the reversal of reduction becomes simple. When a reversal is triggered there is a chain of reversals started as each stage must be reversed individually.

6 Evaluation

This section discusses the final results of the implemented system concerning the requirements discussed in Section 3.

6.1 Overlapping Elements

Significantly fewer elements overlap when compared to the works of Keenan, Citrin et al. and Massalõgin. The issues of application precedence for all three and variable binding for Keenan and Citrin et al. were solved by overlapping elements of the graph. The system designed here avoids this undesirable solution, reducing the mental overhead on the user. The systems of Citrin et al. and Thyer both avoided the issue

The only overlapping elements of the static graph are the borders of expressions. This overlapping is not as adverse to the user's interpretation as the elements are completely overlapped, giving the appearance of a single element rather than individual ones. However, it can be difficul to parse the scope of expressions as can be seen in Figure 6.1 where it is not clear whether the abstraction highlighted extends to point A or B.

This issue is mitigated somewhat by the ability to highlight expressions individually by the user but this is not a perfect solution as it requires user intervention.

There is also the overlapping of expressions in the animation of beta reduction but this is seen as more desirable than elements disappearing and reappearing as discussed in the requirements section.

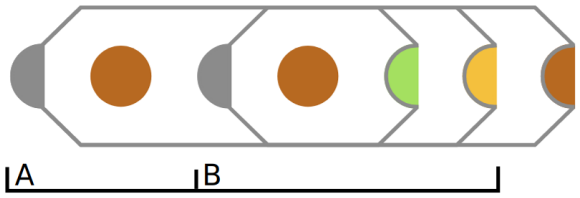

Figure 6.1: Demonstration of scoping issue

6.2 Spatial Limits

Expanding expressions is handling in multiple ways in the system, satisfying each requirement laid out. The use of primitives is the most significant as it can drastically reduce the complexity of the graph on view to the user. Large expressions which are not relevant at the current stage of reduction can be hidden behind primitives allowing the user to focus on pertinent elements.

Users can focus on specific subsets of the expression tree by selecting a node. The graph will change to only child elements of that expression and the expression text displayed in the controls will change to only the text version of the focused expression. This functionality also has the advantage of allowing the user to reduce expressions within the expressions of their choice which can demonstrate the differences in method results.

Finally, the user can easily scale the size of the graph to suit their needs. This is especially useful when reducing large expressions which can expand rapidly beyond the size of the screen in the default scale.

6.3 Distinct Shapes

As the design of the system is most similar to Keenan's, the shapes used for variables and abstractions are distinct, unlike the others. Unfortunately, there is a

clear overlap of the designs for abstractions and applications. Applications having explicit shapes were chosen to be more desirable over the use of connecting lines across the graph to show application. The similar shape was chosen as the boundaries of each are supposed to be similar to the parentheses used in the textual version. Although there is a difference in the presence of an input 'socket' on abstractions, it is certainly possible a better shape could be chosen while preserving the output 'plug'.

Primitives are also of the same shape as abstractions but this choice was deliberate. The solid colour and label provide adequate distinction while a similar shape maintains the sense that they have no distinction in the reduction of expressions.

6.4 Clear Application Order

Conveying application order is an area where the system differs from all previous designs. Each design relied on labels or overlapping elements to show application order. The use of explicit application elements provides a way of showing application order while avoiding these pitfalls. 'Plugs' in nested applications act as barriers to reduction and thus provide a visual cue to the user that the contents of the application must be addressed as a single expression.

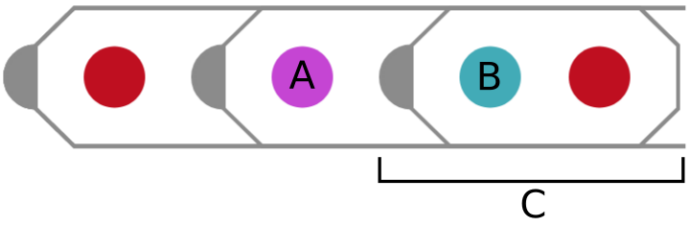

Figure 6.2: Application C shows a clear barrier between variables A and B

6.5 Colour Support

The system uses colours to distinguish variables as in Massalõgin's work. The issue with this is that the colours are randomly generated without any sort of duplicate or similarity checking. Occasionally this is a problem as multiple variables appear with the same colour. Generation of arbitrary numbers of colours is a complex

issue, justifying its own area of research. Some work has been done in the area that provides implementations[28], but the problem was seen as too large to tackle for this project due to time constraints.

The problem is addressed by the ability to highlight the variables bound an abstraction but, similarly to the issue of ambiguous expression scope, this requires user intervention. This also means that there is no way of displaying the variables bound by multiple abstractions at the same time should more than two abstractions be assigned the same colour although this situation is extremely rare. Other solutions such as not using random colours, instead using a cycling set of colours, could be implemented. This was not done as it would add significant complexity and state to the system, since the current implementation is stateless.

6.6 Labelling

The only labelling when viewing the graph passively is within primitives. Labels are used for primitives as using specific shapes for each would both limit the primitives that could be used and greatly increase the complexity of their implementation, making creating custom primitives more difficult for users. This use of labels is seen as acceptable as the number of primitives is generally small and so does not overload the user with information.

Rather than directly labelling other types of expressions, the tooltip allows users to show the textual version of expressions of interest. Although this also requires user intervention as discussed in other solutions, the tooltip is not a problem as the other solutions are because the textual representation is not essential for the understanding of the expressions but instead acts as a supplementary source of information.

6.7 Smooth Motion

Smooth motion was a major factor in the design of the system and the results demonstrate the attention to maintaining success in this area. All aspects of change within the system are smoothly animated. Reduction animation is carried out in a series of coordinated steps so that the user can easily follow along. At no point does any element disappear and reappear at another position.

As important as the inclusion of smooth motion is the ability for the user to adjust the speed of the motion to their liking. This ability is essential for both new and experienced users. New users can gain a better grasp for the workings of the system by slowing the animation down to a comfortable pace, while experienced users can accelerate motion during stages that are of less interest to them and slow it back down once they reach a point they wish to follow more closely.

6.8 Interaction

Interaction is implemented in multiple ways across the system, through the controls and the graph itself.

Every element of the graph can be interacted with in some way and the graph can also be moved around as a whole. These interactions allow users to become comfortable with the system as they explore the information displayed across the graph and tooltip. The graph interactions are also not mandatory for the user and thus do not add significant overhead to using the system from the beginning.

The controls offer another range of interactions to the user but instead are used to change the information on display rather than simply displaying information. These controls offer a way of manipulating expressions that is simple and easy to use. Massalõgin's system had controls in both the graphical interface as well as the command-line interface. This method of control is confusing as it splits the user between two very different forms of interaction. Command-line interfaces are also less intuitive to new users. By maintaining all interaction within the confines of a single web page, the user is constantly kept within a single context which reduces mental overhead.

7 Conclusion

7.1 Overview

This work attempted to provide a design for a system that could answer how could a set of rules be formalised to allow any arbitrary lambda expressions to be drawn, and how can a series of steps showing beta reduction be animated to allow for arbitrary expressions. The work also attempted to improve upon the existing designs answering similar questions and provide an effective tool for communicating information about the Lambda Calculus.

The system provided allows for any lambda expression to be drawn in a visual representation and beta reduced using animated steps, with no limits inherently brought upon by the design. Data visualisation standards were followed in the design to ensure effective visual communication, although not all issues present in previous works were solved to complete satisfaction.

This work provides both a design for a system and implementation of such a system to be used for Lambda Calculus visualisation. This system acts as the basis for a potential learning tool, aimed at, but not limited to, use within educational institutions.

7.2 Future Work

As this work is meant to be a starting point for further development and evaluation, there is a large scope for future work.

Visualisation Evaluation

Further evaluation of the system would be necessary before being employed in educational contexts. Unfortunately due to global circumstances and time constraints at the time of writing this paper, this was not possible and so is left to future work. As discussed in the evaluation, there are some issues present in the design of the static expressions. These would need to be addressed before the visualisation is ready to be evaluated further.

An effective method of evaluation would be traditional user trials, involving users with and without experience of the Lambda Calculus. Multiple types of trials could be performed, to test both the design and implementation.

The most essential evaluation necessary would be measuring its effectiveness in education of the Lambda Calculus. People unfamiliar with the Lambda Calculus would of course be necessary test subjects, but they could be varied in their experience of other relevant areas such as mathematics or technology. From the beginning, the design was created to be easy to both understand and use. It would certainly be useful to gain insight into the ability of users who had little to no relevant experience in understanding the system. To perform such tests, the knowledge of those taught using the system could be compared to those taught using traditional teaching methods. These methods could include teaching using the purely textual form and then also a graphical approach more commonly used, such as the tree visualisation in Thyer's work. Differences in preference of learning method might be found in users based on their existing experience.

To test the effectiveness of the implementation, there are a number of activities users could participate in. Initially they could observe the system perform a number of simple reductions and then more complex ones, followed by a number of questions to assess their understanding. Following this they could be asked to create their own expressions to evaluate. As they go about operating the system, notes would be taken of any questions or misunderstandings they have. Common issues could then be addressed and the process repeated until an acceptable level of usability is reached. The exact level that is accepted will have to be decided upon, based upon the time and resources available.

Feature Expansion

There are some features that could be added to improve the design and implementation.

The ability for users to create custom built-in functions and save them to the system would be useful as they could utilise these in multiple expressions without needing to remember the expressions. There could be an option to save the current expression at any point along reduction.

Another feature to improve general user experience would be the ability to set a max depth for visualisation. Performance becomes an issue when there a large number of expressions present. If a user could limit the number of expressions shown using a depth limit, this problem would be significantly reduced. There would need to be a way gracefully handling the changing of expressions from being hidden or shown as expressions are reduced, while also not reducing the efficacy of the visualisation at showing the expression.

Reduction Methods

The reduction methods provided in this work are simple and only provided for demonstration. Several improvements could be made to the reduction system.

Many reduction methods are derivations of what is generally referred to as 'call by need' where applications do not duplicate argument expressions but instead maintain multiple references to a single source. When any of these references is needed to be reduced, the other references are also reduced, eliminating duplicated work. This is often called 'lazy evaluation' and is prevalent in the world of Lambda Calculus. This work does not accommodate these types of reduction methods but could be altered to allow for them.

Once this kind of reduction is possible, there is a possibility for use of the system outside of educational settings. If more complex methods were possible and users could explore them visually, there could be a case made for the use of this application in research. A potentially viable path for future work would be allowing users to implement custom strategies and watch them used in reduction. The current implementation is already designed in a way to allow for pluggable reduction methods but there would need to be a suitable API developed so users could easily add their own versions.

Bibliography

[1] A. M. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2 (4):153163, 1937. doi: 10.2307/2268280.

[2] Haskell B. Curry. *Combinatory logic*. Studies in logic and the foundations of mathematics. North-Holland Pub. Co. ISBN 978-0-7204-2208-5.

[3] Stack overflow developer survey 2019, . URL `https://insights.stackoverflow.com/survey/2019/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2019`.

[4] Achim Jung. A short introduction to the lambda calculus. *School of Computer Science, The University of Birmingham*, 2004.

[5] Marat Boshernitsan and Michael S. Downes. Visual programming languages: a survey. Technical Report UCB/CSD-04-1368, EECS Department, University of California, Berkeley, Dec 2004. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2004/6201.html`.

[6] Scratch - imagine, program, share, . URL `https://scratch.mit.edu/`.

[7] To dissect a mockingbird: A graphical notation for the lambda calculus with animated reduction, . URL `http://dkeenan.com/Lambda/`.

[8] Raul Rojas. A tutorial introduction to the lambda calculus. page 9.

[9] Peter Sestoft. Demonstrating lambda calculus reduction. page 16.

[10] H. P. Barendregt. *The lambda calculus: its syntax and semantics*. Number v. 103 in Studies in logic and the foundations of mathematics. North-Holland ; Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, rev. ed edition. ISBN 978-0-444-86748-3 978-0-444-87508-2.

[11] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of*

programming languages - POPL '90, pages 16–30. ACM Press. ISBN 978-0-89791-343-0. doi: 10.1145/96709.96711. URL http://portal.acm.org/citation.cfm?doid=96709.96711.

[12] Alonzo Church. An unsolvable problem of elementary number theory. 58(2): 345. ISSN 00029327. doi: 10.2307/2371045. URL https://www.jstor.org/stable/2371045?origin=crossref.

[13] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. 2(3): 345–364. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796800000423. URL https://www.cambridge.org/core/product/identifier/S0956796800000423/type/journal_article.

[14] Raymond M Smullyan. *To Mock a Mockingbird: and other logic puzzles including an amazing adventure in combinatory logic*. Oxford University Press, USA.

[15] Programming with visual expressions, . URL http://users.encs.concordia.ca/~haarslev/vl95www/html-papers/citrin/citrin.html.

[16] Viktor Massalõgin. Visual lambda calculus. *Master's thesis, Rijksuniversiteit Groningen, Estonia*, 2008.

[17] Lambda animator : animated reduction of the lambda calculus, . URL http://thyer.name/lambda-animator/.

[18] Matthew Ward, Georges G. Grinstein, and Daniel Keim. *Interactive data visualization: foundations, techniques, and applications*. CRC Press, Taylor & Francis Group, second edition edition. ISBN 978-1-4822-5737-3.

[19] Sidney L Smith and Jane N Mosier. Guidelines for designing user interface software. Technical report, Citeseer, 1986.

[20] Ji Soo Yi, Youn ah Kang, and John Stasko. Toward a deeper understanding of the role of interaction in information visualization. 13(6):1224–1231. ISSN 1077-2626. doi: 10.1109/TVCG.2007.70515. URL http://ieeexplore.ieee.org/document/4376144/.

[21] React a JavaScript library for building user interfaces, . URL https://reactjs.org/.

[22] Framer motion, . URL https://framer.com/motion/. Library Catalog: www.framer.com.

[23] Redux - a predictable state container for JavaScript apps., . URL `https://redux.js.org/`.

[24] Bulma: Free, open source, and modern CSS framework based on flexbox, . URL `https://bulma.io`.

[25] Material-UI: A popular react UI framework, . URL `https://material-ui.com/`.

[26] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. North-Holland, 1972.

[27] jneen/parsimmon: A monadic LL(infinity) parser combinator library for javascript, . URL `https://github.com/jneen/parsimmon/`.

[28] Paola Campadelli, Roberto Posenato, and Raimondo Schettini. An algorithm for the selection of high-contrast color sets. *Color Research & Application: Endorsed by Inter-Society Color Council, The Colour Group (Great Britain), Canadian Society for Color, Color Science Association of Japan, Dutch Society for the Study of Color, The Swedish Colour Centre Foundation, Colour Society of Australia, Centre Français de la Couleur*, 24(2):132–138, 1999.

Appendix

Source code for the implementation, at the time of writing, is hosted at https://github.com/rcarriga/viscal. A working version of the implementation is also available for use at https://rcarriga.github.io/viscal.