UNIVERSITY OF BIRMINGHAM

A visualiser for linear λ-terms as 3-valent rooted maps

George Kaye
1522391

Supervisor: Noam Zeilberger

School of Computer Science
MSci in Computer Science

Contents

1 Abstract

This report details the development of a set of tools in Javascript to aid in the research of the topological properties of linear λ-terms when they are represented as 3-valent rooted maps. A λ-term visualiser was developed to visualise a λ-term specified by the user as a rooted map on the screen. The visualiser also includes functionality related to normalisation of terms, such as the option to view a normalisation graph or reduce a term to its normal form. To complement this a λ-term gallery was created to generate λ-terms that satisfied criteria specified by the user, and display their corresponding maps. While the focus of the project was on linear λ-terms, these tools also work for all pure λ-terms. The tools can be used for a variety of different applications, such as examining the structure of different terms, disproving conjectures regarding various subsets of the λ-calculus, or investigating special normalisation properties held by different sets of λ-terms. We evaluate the tools' success and acknowledge that while the tools suffer from performance issues when used for larger terms, they still fulfil many of the original aims of the project, and may still be very useful for systematic exploration of the λ-calculus in the future.

Keywords: *λ-calculus, normalisation, rooted maps, topology, combinatorics*

Note on the content of this report

Some of this report has been based on the content written in my Scientific Paper (Kaye, 2018b).

Source code

The source code for this project can be found at `https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2018/gjk591`.

2 Acknowledgements

Thanks to my supervisor, Noam Zeilberger, for his guidance and support throughout this project, especially by introducing many of the background concepts required. Thanks also to my friends at university and my family at home, for keeping me going throughout the year.

3 Introduction

3.1 Motivation

There are many links between the λ-calculus and areas of mathematics and computer science. One such area is graph theory – we can draw λ-terms as graphs on surfaces, which are also known as maps. We can then observe how the topological properties of these maps link with the computational and logical properties of the original terms.

The idea of representing general λ-terms as graphs is not new - Wadsworth (1971) and Statman (1974) studied it for their PhD theses. One of the most common ways of representing λ-terms is by an abstract syntax tree (Erwig, 1998), with nodes representing abstractions and applications – an example of this is shown in Figure 1. Numerous other ways of representing λ-terms have been developed, using nested structures (Citrin et al., 1995), bubbles (Massalõgin, 2008) or even as a game involving alligators (Victor, 2007)!

It is only more recently that links have been uncovered between the *linear* λ-calculus and the combinatorics of rooted maps (Bodini et al., 2013). Every linear λ-term corresponds to a unique rooted map, and the number of maps that exist for different families of λ terms can form known sequences (Zeilberger, 2016). It can be interesting to draw out these different maps and examine the topological properties shared between λ-terms.

However, performing experimental mathematics with λ-terms and their maps can be a time-consuming process. While there are several examples of software developed for visualising λ-terms (e.g. Bharadwaj (2017), Massalõgin (2008), Thyer (2007)), there do not appear to have been any implementations that attempt to represent λ-terms as their corresponding rooted maps. Therefore, the main motivation of this project was to develop tools that could be used to generate these maps, reducing the time needed to draw these maps so that more time can be spent on actual research.

There were two tools developed: a **λ-term visualiser** and a **λ-term gallery**.

3.2 λ-term visualiser

The first tool can generate maps for λ-terms from user input, in addition to calculating interesting properties such as crossings. An example of the visualiser in use can be seen in Figure 2. By visualising the λ-terms it can become much easier to understand the structure of more complex structures implemented in the λ-calculus (such as pairs). While linear λ-terms were the main focus during development, steps were taken to ensure that any pure λ-term could be represented in some way, so as not to reduce the applications of the tool.

The visualiser also has functionality relating to the normalisation of terms. The β-redexes contained within the term are listed, and by clicking on these the user can reduce the term to its normal form (if one exists). A normalisation graph can also

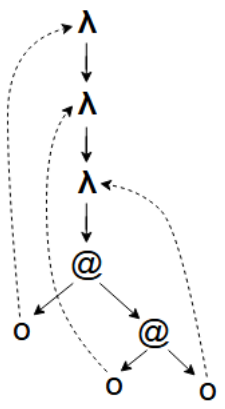


Figure 1: A representation of the term $\lambda x.\lambda y.\lambda z.x\,(y\,z)$ as an abstract syntax trees, with pointers representing the use of abstracted variables.

be generated, which can be useful when investigating how normalisation properties, such as complexity, differ between different subsets of the λ-calculus.

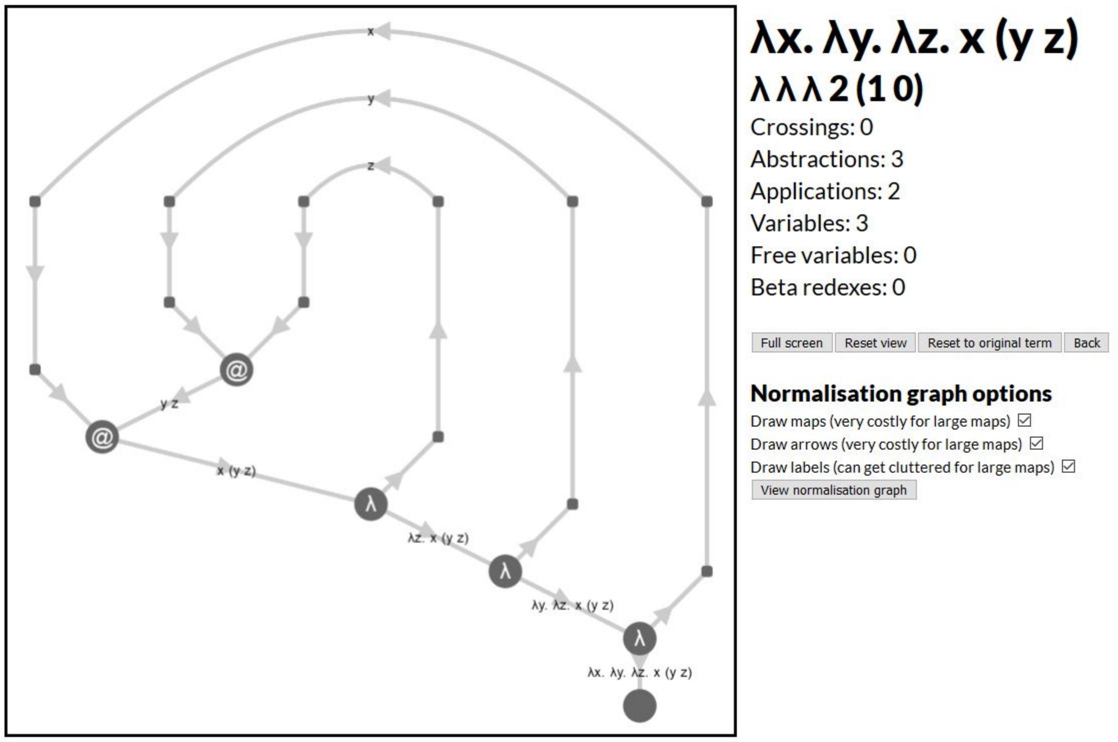
3.3 λ-term gallery

When studying properties of λ-terms, it can be useful to generate terms and look for interesting properties shared between terms and their maps. However it can be tricky to come up with terms with certain properties (such as terms containing a certain amount of subterms) on the fly. The λ-term gallery can generate all terms of a certain size and free variables, with the ability to filter based on properties such as crossings or β-redexes. This makes disproving conjectures by finding counterexamples a much easier process. An example of a λ-term gallery can be seen in Figure 3.

From the gallery, the user can inspect the generated terms using the visualiser, with the same functionality present as detailed above.

3.4 Report structure

The rest of the report is structured as follows. Section 4 provides the necessary background information for the project. Section 5 details the requirements for the tools to be a success, and Section 6 details the implementation choices that went into developing the tools to fit this specification. Section 7 covers the features in the final version of the tools along with some examples of how they could be used and Section 8 details the testing strategy used. Section 9 evaluates the success of the project, and Section 10 details how the project was managed. Appendix A details the file structure of the submitted ZIP file and Appendix B summarises the logs detailing the status of the project each week.



λx. λy. λz. x (y z)

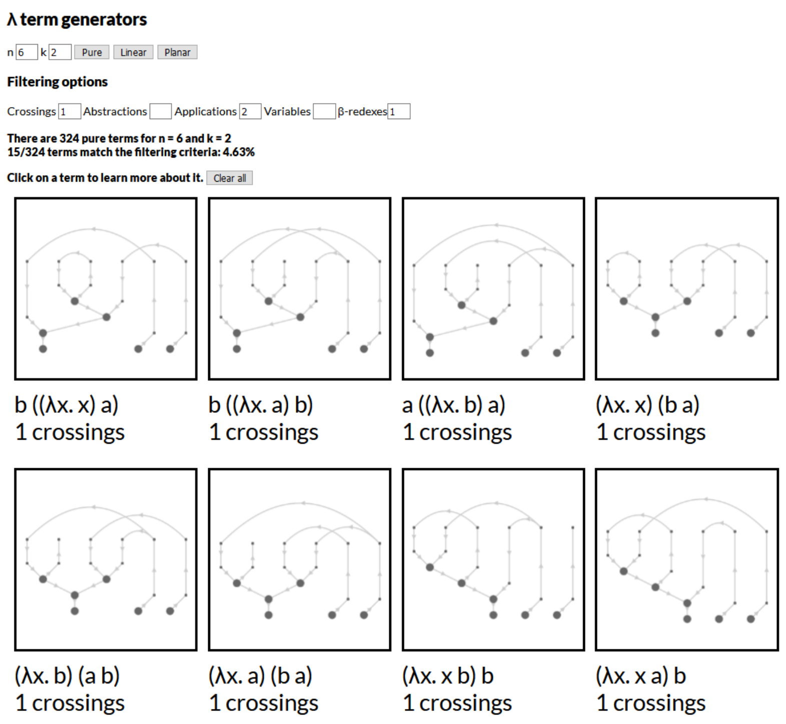
λ λ λ 2 (1 0)

Crossings: 0
Abstractions: 3
Applications: 2
Variables: 3
Free variables: 0
Beta redexes: 0

Normalisation graph options

Draw maps (very costly for large maps) ☑
Draw arrows (very costly for large maps) ☑
Draw labels (can get cluttered for large maps) ☑

Figure 2: The λ-term visualiser, representing the term from Figure 1 as a 3-valent rooted map, along with some properties of the map on the right.



λ term generators

n 6 k 2 Pure Linear Planar

Filtering options

Crossings 1 Abstractions Applications 2 Variables β-redexes 1

There are 324 pure terms for n = 6 and k = 2
15/324 terms match the filtering criteria: 4.63%

Click on a term to learn more about it. Clear all

b ((λx. x) a)
1 crossings

b ((λx. a) b)
1 crossings

a ((λx. b) a)
1 crossings

(λx. x) (b a)
1 crossings

(λx. b) (a b)
1 crossings

(λx. a) (b a)
1 crossings

(λx. x b) b
1 crossings

(λx. x a) b
1 crossings

Figure 3: The λ-term gallery, displaying all closed linear terms of size 5.

4 Background

This section will cover some concepts and terminology that will be used in the remainder of the report.

4.1 The λ-calculus

The λ-calculus is a model of computation where programs are represented by variable abstraction and function application. It is the basis of all functional programming languages.

4.1.1 Definitions

The simplest terms in the λ-calculus (λ-terms) are just **variables** ($x, y, z, \ldots$). More complex terms can be created using the operations of **abstraction** ($\lambda x.t$) and **application** ($t_1 t_2$). For clearer notation, applications are left-associative and abstractions extend as far to the right as possible:

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

Variables in the λ-calculus can be **bound** or **free**. A variable is bound if it is inside the scope of a corresponding λ-abstraction (i.e. it is a local variable), or free otherwise. For example, in $\lambda x.x\,y$, the x is bound but the y is free. A λ-term with no free variables is called a **closed term**. Two λ-terms are said to be **α-equivalent** if the only difference between them is the names of their bound variables – for example, $\lambda x.x$ and $\lambda y.y$ are α-equivalent. The process of renaming bound variables is known as **α-conversion**:

$$\lambda x.t \rightarrow_\alpha \lambda y.t[x \mapsto y]$$

To avoid ambiguity between α-equivalent terms, we can use **de Bruijn notation**. Rather than using explicit variable names, each variable is instead represented by how 'far away' the corresponding abstraction is – how many lambdas one has to pass through to find the right one. For example, $\lambda x.\lambda y.\lambda z.x\,y\,z$ can be written as $\lambda\,\lambda\,\lambda\,2\,1\,0$. This eliminates the need for α-conversion, and makes it much easier to implement checking for equality between λ-terms.

λ-terms contain a number of **subterms**, defined as:

$$\begin{aligned} \mathit{subterms}(x) &= 1 \\ \mathit{subterms}(\lambda x.t) &= 1 + \mathit{subterms}(t) \\ \mathit{subterms}(t_1 t_2) &= 1 + \mathit{subterms}(t_1) \ + \ \mathit{subterms}(t_2) \end{aligned}$$

4.1.2 β-reduction

Program execution in the λ-calculus is performed through **β-reduction** – applying functions to their arguments. A term of the form $(\lambda x.t)\,u$ is called a **β-redex** and can be β-reduced as follows:

$$(\lambda x.t)\,u \rightarrow_\beta t[x \mapsto u]$$

Repeatedly performing β-reduction on a term until it contains no β-redexes is known as **normalisation**. A term with no β-redexes is in its **normal form**. The normal form of a λ-term is necessarily unique if it exists – the order that β-redexes are chosen does not matter. This is known as the **Church-Rosser theorem** (Church and Rosser, 1936). However computing if normalising a pure term will terminate is undecidable (Church, 1936). This is because attempts to reduce a λ-term may lead to a loop, or just continuous expansion. One well-known example is the Ω term:

$$\Omega = (\lambda x.x\,x)(\lambda x.x\,x) \rightarrow_\beta (x\,x)[x \mapsto (\lambda x.x\,x)] \equiv (\lambda x.x\,x)(\lambda x.x\,x) \equiv \Omega$$

When it is possible to get stuck in one of these normalisation loops, the order of reduction can actually matter. This is shown in the example below, where reducing redex 1 leads to a new term whereas redex 2 leads to the same term:

$$T = \underbrace{(\lambda x.\lambda y.x)\,a}_{\text{redex 1}}\ \underbrace{((\lambda x.x\,x)(\lambda x.x\,x))}_{\text{redex 2}}$$

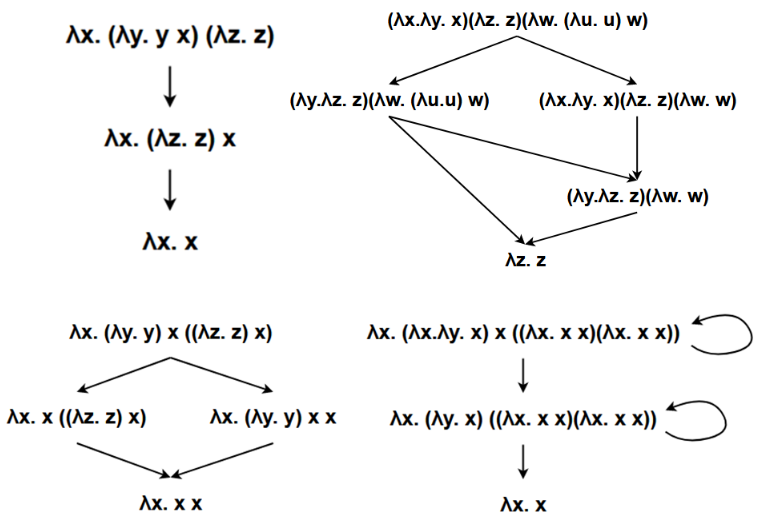


Figure 4: Several examples of normalisation graphs, showing how different terms can have paths of different lengths, diverging paths or normalisation loops.

$$T \to_{\beta 1} (\lambda y.a)((\lambda x.x\,x)(\lambda x.x\,x)) \to_\beta a$$
$$T \to_{\beta 2} (\lambda x.\lambda y.x)\,a\,((\lambda x.x\,x)(\lambda x.x\,x)) \equiv T$$

The choice of which redex to reduce is related to **evaluation strategies**. Choosing the **outermost reduction** corresponds to **normal order** evaluation, in which arguments are substituted into a function before they are evaluated. Conversely, choosing the **innermost reduction** corresponds to **applicative order** evaluation, where arguments are fully evaluated before being applied to functions. Always choosing the outermost reduction is guaranteed to find the normal form if it exists. This is not the case for choosing the innermost reduction since an argument that is not used in a function could contain a normalisation loop.

The process of normalisation can be represented by **normalisation graphs**, which show the various paths between a λ-term and its normal form. Several normalisation graphs can be seen in Figure 4.

4.1.3 Fragments of the λ-calculus

The **pure λ-calculus** contains all terms formed from combining variables, abstractions and applications. However we can restrict ourselves to smaller **fragments** of the λ-calculus. The **linear λ-calculus** is a subset of the pure λ-calculus containing terms in which variables are used exactly once – this is useful when considering scenarios involving resource management. The **planar λ-calculus** is a subset of the linear

λ-calculus in which variables are used in the order they are abstracted in. Examples of terms from these different fragments are shown in Table 1.

Linear and planar terms have special properties relating to normalisation. Linearity and planarity are preserved by β-reduction, and all linear and planar λ-terms have a computable normal form. This is because as each variable in a linear term only occurs once, terms can only get smaller with each β-reduction – they cannot 'blow up'. Similarly, all paths to the normal form are the same length – because all abstracted variables *must* be used, there are no 'shortcuts' to the normal form by choosing a specific redex.

It can be shown that the normal form of a linear λ-term is computable in polynomial time. To normalise a linear term, a search must be performed through the term to try and find a β-redex $(\lambda x.t)\,u$. Inside this β-redex, there will only need to be one substitution – replacing the single occurrence of the abstracted variable x in t with the applied subterm u. Using the definitions of $subterms(t)$:

$$subterms((\lambda x.t)u) = 1 + (1 + subterms(t)) + subterms(u)) \tag{1}$$

$$subterms(t[x \mapsto u]) = subterms(t) - 1 + subterms(u) \tag{2}$$

By subtracting 2 from 1, we can see that the number of subterms always shrinks by three in a β-reduction. Therefore, the most reductions that a linear λ-term can have before reaching a normal form (i.e. if the term reduces to a lone variable with one subterm) is $\frac{n}{3}$. In the 'worst case' scenario, a program attempting to normalise a linear λ-term would have to normalise n terms, taking $\frac{n}{3}$ operations each time. So the upper bound of complexity of normalising linear λ-terms in in $\mathcal{O}(n^2)$ – polynomial time.

It has also been shown by Mairson (2004) that normalising linear λ-terms is PTIME-complete, by encoding boolean circuits as λ-terms. By reducing the Circuit Value Problem (known to be PTIME-complete) to normalising linear λ-terms, this proves that the latter is also PTIME-complete.

Since all planar terms are also linear terms, they share this upper bound of normalisation complexity. However, there may be a lower bound of complexity that is unknown to us. We cannot use Mairson's boolean circuits to prove this since the functions are not planar. Studying the normalisation properties of planar terms is a possible use for the tools developed in this project, perhaps by examining normalisation graphs.

4.2 Graphs and maps

In graph theory, a **graph** is a set of nodes and edges that link pairs of nodes. When these graphs are **embedded** onto a surface they are called **maps**. Unlike graphs, the order of edges around a node is important for maps, and the same graph can be represented as many different maps (an example is shown in Figure 5). A map has a **genus** which is how many 'holes' the surface it is embedded into has. **Planar**

Term	Pure	Linear	Planar
$\lambda x.x$	Yes	Yes	Yes
$\lambda x.(\lambda y.y)\,x$	Yes	Yes	Yes
$\lambda x.\lambda y.x\,y$	Yes	Yes	Yes
$\lambda x.\lambda y.y\,x$	Yes	Yes	No
$\lambda x.x\,x$	Yes	No	No
$\lambda x.\lambda y.x$	Yes	No	No

Table 1: Examples of terms in various fragments of the λ-calculus.

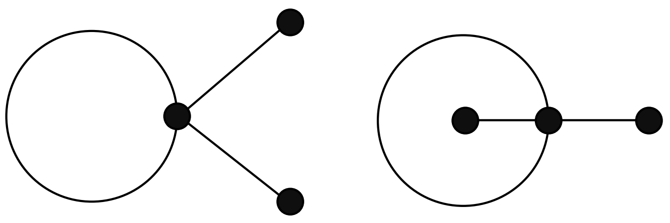


Figure 5: These two diagrams represent the same graph but two distinct maps (the ordering of edges around the point on the circle is changed). Adapted from Lando and Zvonkin (2013).

maps are maps with no crossings of edges – they have a genus of 0. A map can be represented as a set of nodes, half-edges, and permutations representing the order of half-edges around nodes, and pairs of half-edges that form edges. This representation is known as a **combinatorial map**, and more about them can be found in Zeilberger (2016).

In this project we are particularly interested with **rooted 3-valent maps**. The **valency** of a node is how many edges connect to it - maps where all of the nodes have a valency of 3 are called **3-valent**. We can make a **rooted map** by adding a 'special' node (the **root**) that connects to the map at one point, such as in the example in Figure 7.

We can represent λ-terms as maps, with abstractions and applications represented by nodes, as shown in Figure 8. We can think of the ordering of edges around nodes

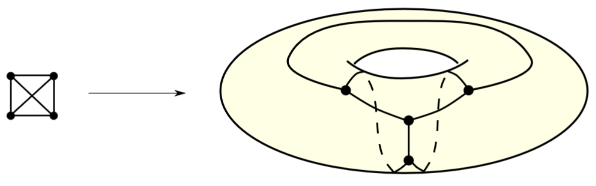


Figure 6: An example of how a graph with crossings can be embedded onto a torus. From Zeilberger (2018).

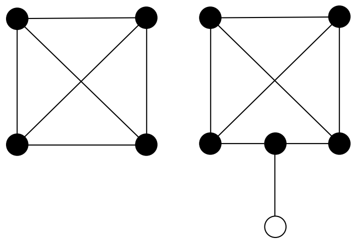


Figure 7: A 3-valent map, and the same map but rooted (root indicated by the white node)

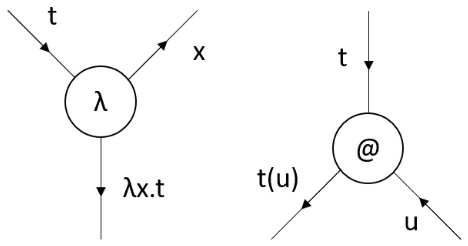


Figure 8: An abstraction and an application, represented as nodes of a map.

in term of their types.

For an abstraction node the edges flow anti-clockwise:

- The full abstraction $\lambda x.t$ flowing out `:: (A -> B)`
- The abstracted variable x flowing out `:: A`
- The body of the abstraction t flowing in `:: B`

For an application node the edges flow clockwise:

- The function t flowing in `:: (A -> B)`
- The argument u flowing in `:: A`
- The application $t(u)$ flowing out `:: B`

With the addition of a root to represent the start of the term, these nodes can be combined to create a rooted map, as shown in Figure 9.

There are several bijections between fragments of the λ-calculus and families of maps (Zeilberger, 2016). One already mentioned is the bijection between closed linear λ-terms and rooted 3-valent maps. Rooted planar maps also form a bijection with closed planar λ-terms.

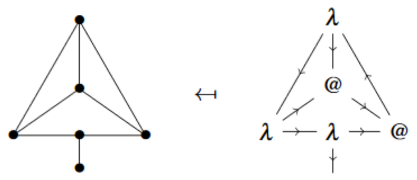


Figure 9: A representation of the term $\lambda x.\lambda y.\lambda z.x\,(y\,z)$ as a rooted 3-valent map, without and with node labels. From Zeilberger (2016).

5 Specification

This section details the requirements for the tools to be a success, using the three main parts of the project detailed in the Proposal (Kaye, 2018a).

5.1 A graphical representation of λ-terms

5.1.1 Parsing terms from user input

The tools must be able to parse user input into λ-terms.

Correctness All valid user input must be parsed into the correct λ-term.

Reliability The parser should not crash upon receiving bad input (e.g. mismatched brackets), and should instead return some sort of parse error.

5.1.2 Visualising terms

The tools must be able to generate the corresponding rooted map for a λ-term specified by the user.

Correctness The generated maps must accurately represent the λ-term entered by the user. The ordering of edges around nodes is the most important point to consider here since different ordering of edges can lead to completely different maps.

Clarity The generated maps must be easy to understand by the user. This means that the elements shouldn't be cramped together but laid out clearly on the screen so that it is easy to identify the individual parts of the map.

Consistency The maps should be generated in such a way that similar terms generate similar maps – there should be a consistent structure throughout. This means that it is easy to observe the differences between terms since they will stand out.

Performance The maps should be generated in a suitable time.

Completeness The visualiser must be able to fulfil these criteria for *all* λ-terms, whether pure, linear or planar. This will mean that the visualiser can be used for many different applications, not just for investigating a particular subset of the λ-calculus.

Aesthetics A somewhat minor point, but it would be a bonus if the generated maps were pleasing to look at, so the user feels compelled to generate more and keep up their research.

5.2 Generating λ-terms from a given fragment

The tools must be able to generate galleries of λ-terms that fulfil specified criteria.

Completeness The generator must return *all* terms that match the criteria specified by the user.

Soundness All terms generated must satisfy the criteria specified by the user.

Clarity The generated terms should be set out in a way that is easy for the user to understand (e.g. laid out in a grid on the screen).

Performance The λ-term galleries should be generated in a suitable amount of time.

5.3 Studying normalisation properties of fragments

The tools must be able to display the β-redexes contained within a term, perform these redexes, and generate the normalisation graphs for terms.

Correctness Any performed β-redexes must lead to the correct term. Generated normalisation graphs must display the correct edges leading to the correct nodes, ensuring that redexes that lead to the same term also point to the same node on the graph.

Completeness All β-redexes in a term must be found and displayed. The normalisation functionality must be usable on all terms in the same way, regardless of the number or location of β-redexes. For terms with infinite reduction graphs, a portion of the graph should be shown.

Performance The normalisation graphs should be generated in a suitable amount of time.

6 Implementation

This section will cover the implementation of the tools, and issues that rose.

6.1 Language

To implement the tools, I decided to use Javascript. This was so the tools could be distributed as 'web apps' and hosted on a website rather than having to be downloaded and relying on any dependencies.

6.2 Implementing the λ-calculus in Javascript

The first step in the project was to implement the λ-calculus in Javascript, as a basis for the rest of the project to build on. The implementation was partially based on the ML examples developed by Pierce (2002). The three types of λ-term (variables, applications and abstractions) are implemented as Javascript classes. Variables are stored as de Bruijn indices – this means that it is trivial to tell if terms are identical without having to perform α-conversion. Abstractions and applications are constructed as combinations of subterms, so complete λ-terms are represented as trees, where nodes may have zero (for variables), one (for abstractions) or two (for applications) children. Functionality was added later in the project so that terms can be associated with an **alias** to save time when writing out large expressions (for example, the alias `id` for the identity function $\lambda x.x$). An example of how terms can be represented can be seen in Figure 10.

Since terms often contain free variables, a class to represent the context of a term was also created. This was effectively a wrapper for an array that contained the labels of terms currently in the context, with some extra methods to make manipulating it easier, such as determining a label from a given de Bruijn index.

Printing the λ-terms proved to be more nuanced than expected. The de Bruijn representation of a term is constant and easy to print by traversing the λ-term tree recursively. However it is not very readable (especially in large terms) – traditional labelled terms are much more intuitive. Originally variables stored a 'label' that could be printed instead of the index, but this proved to be quite buggy and often variables and their corresponding abstractions would display different labels (e.g. after α-conversion). To fix this, labels were restricted to just abstractions, and when printing these labels would be added to a context. When the variables were to be printed, the index of the term would be looked up in the context and the appropriate label retrieved. This would ensure consistency (and subsequently correctness) throughout the term.

When implementing the normalisation functionality, it became apparent that some sort of α-conversion would still need to be implemented to avoid clashes of variables in the printed labels. A function was implemented to generate a 'canonical' set of labels,

```
    const y = new LambdaVariable(0);
    const x = new LambdaVariable(1);
    const app = new LambdaApplication(y, x);
    const abs1 = new LambdaAbstraction(app, 'y');
    const abs2 = new LambdaAbstraction(abs, 'x', 'swap');

    abs2.prettyPrint(); // prints \ \ 0 1
```

Figure 10: How the function $swap = \lambda x.\lambda y.y\,x$ can be constructed in the JAVASCRIPT implementation.

to ensure that each variable name was only used once in the term. This worked by renaming all free variables in the context to $a, b, ...$, and then traversing the term and replacing the label associated with each abstraction with a new one from $x, y...$. For example, the term $(\lambda d.(\lambda h.h\,d)\,h)\,g$ with free variables h and g would be α-converted to $\lambda x.(\lambda y.y\,x)\,a)\,b$. This would also be used when generating normalisation graphs later on – redexes that led to α-equivalent terms would have a consistent set of labels rather than juggling many different representations. For example, both redexes in $(\lambda x.(\lambda y.y)\,x)(\lambda z.z)$ reduce to the same term in de Bruijn notation $(\lambda\,0)(\lambda\,0)$, but with different labels. In the normalisation graph, the term is α-converted to its 'canonical' labelling.

6.3 Parsing terms from user input

Initially the parser would iterate through the user input one character at a time, making note of 'special' characters (e.g. a backslash to represent a λ-abstraction, or an opening bracket to indicate the start of a subterm). It would then create the λ-term objects as it went from left to right. However the original parsing algorithm grew quite confusing, as it had to keep track of many different states (e.g. if an abstraction was in progress), and checking for syntax errors had to occur in many different places.

To make the process more less convoluted, parsing was split into two distinct parts, Firstly, the the input would be split into different tokens (e.g. $\lambda var.(\lambda y.y)\,var \implies$ `[\,var,(,\,y,),var)]`) by iterating over characters in the input, creating a new token when encountering a special character or a space. The second part was the actual parsing phase where these tokens would be formed into actual λ-terms. The benefit of tokenising first is that syntax errors (such as mismatched brackets or missing abstraction variables) are caught during this phase, and the parser can iterate over tokens without having to worry about malformed terms. This also simplifies dealing with variables longer than one character, since they can be stored as one token.

This means that the only 'special characters' the parser needs to check for are

`(` and `)` for subterms, and `\` for abstractions. Everything else is a variable, and adjacent tokens/subterms represent applications. When an abstraction or subterm is encountered, their scope is determined by finding the appropriate closing bracket (or the end of the term), and the parse function called recursively on the tokens within the scope. This continues until no tokens are left. To extend the parser to handle aliases, all that had to be added was to check tokens against the list of existing aliases, and insert the corresponding function body if one existed, rather than treating the token as a new variable.

6.4 Drawing the terms

To create the elements in the maps, the λ term tree is traversed recursively. New node objects are created when encountering an abstraction or application, with an edge leading to the previous parent node. When an abstraction is found, the `id` of the abstraction node is stored in a context. When a variable is found, this context is searched to find the corresponding abstraction node – an edge is then drawn between this node and application node where the variable is used. The array of map elements is then passed to the Cytoscape API (`http://js.cytoscape.org/`) which generates the map.

Developing a suitable way of drawing the λ-term maps was the first major problem in the project. Drawing correct maps by hand is quite intuitive as one can place nodes and their edges 'on the fly' so that they do not cross over. However implementing an algorithm for a computer to generate these maps is significantly more difficult. Algorithms that work for some maps may not work for others, so a strategy that generates consistent maps is required. Figures 11 and 12 show how the drawn map for two different terms evolved over time.

Initially maps were generated using a default layout provided by the graph drawing API (1). This placed nodes in a circle and drew the edges as the shortest path between pairs of nodes. While this representation looked tidy, it did not preserve the cyclical order of edges around nodes, so the generated maps were not correct. This was due to the edges representing the use of abstracted variables 'cutting across' the map rather than exiting nodes at the right position. This caused crossings to occur for planar term maps (such as in Figure 11.1). Since edges were always straight unless there were duplicate edges between two nodes, incorrect crossings would also be generated when an edge should have curved around a node to 'dodge' an edge but instead cut straight through it. It was clear from this method that node positions would have to be explicitly set to ensure the correctness of these maps.

In the next algorithm (2), nodes were placed progressively further up the page, with the root at the bottom. To preserve the cyclical order of edges, the scope of abstractions would always be positioned to the left of an abstraction node, and the left and right hand sides of application would be placed to the left and right respectively. To ensure that variable edges would also preserve this ordering, an extra 'support

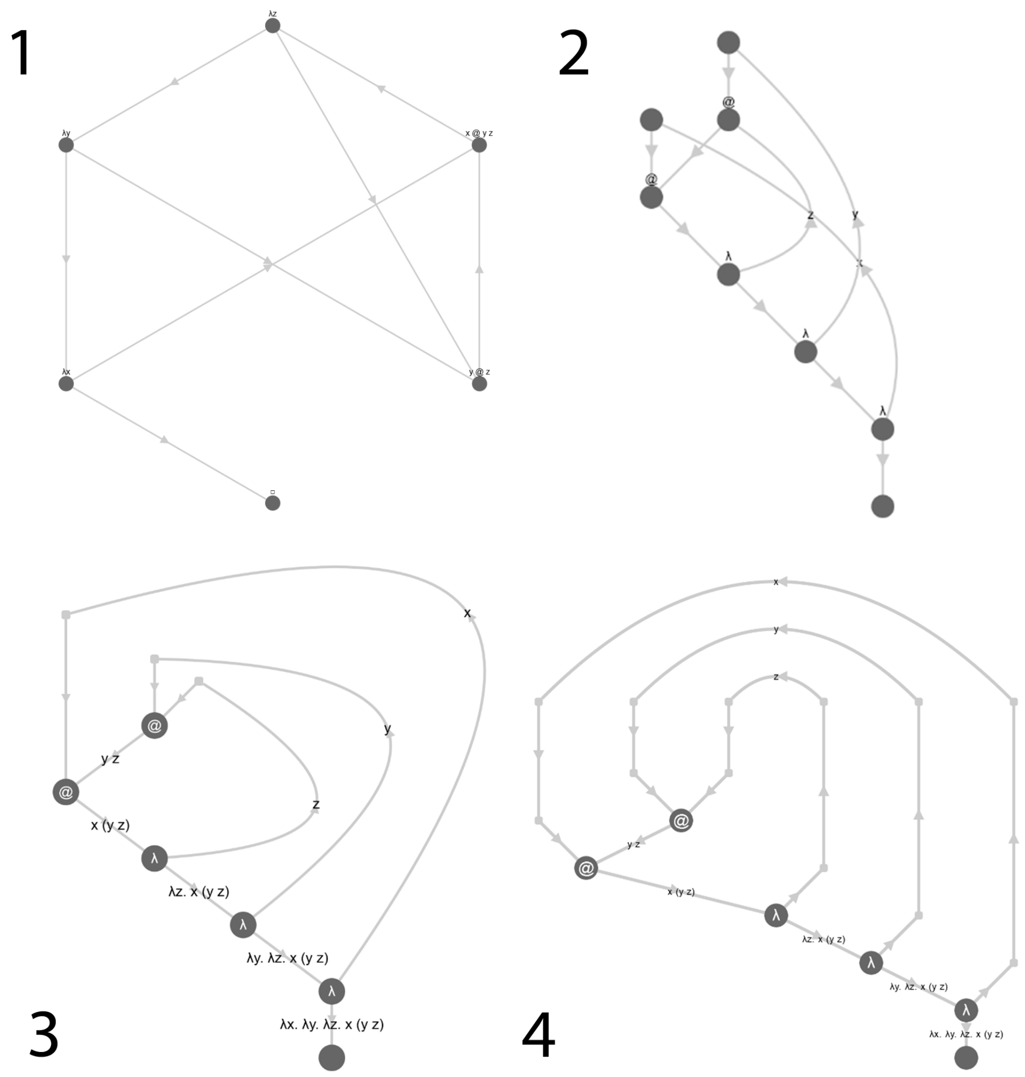


Figure 11: How the map for the term $\lambda x.\lambda y.\lambda z.x\,(y\,z)$ evolved over time. Maps 1 and 2 are incorrect due to too many crossings, Map 3 is correct but not aesthetically pleasing, Map 4 is the final (correct) version.

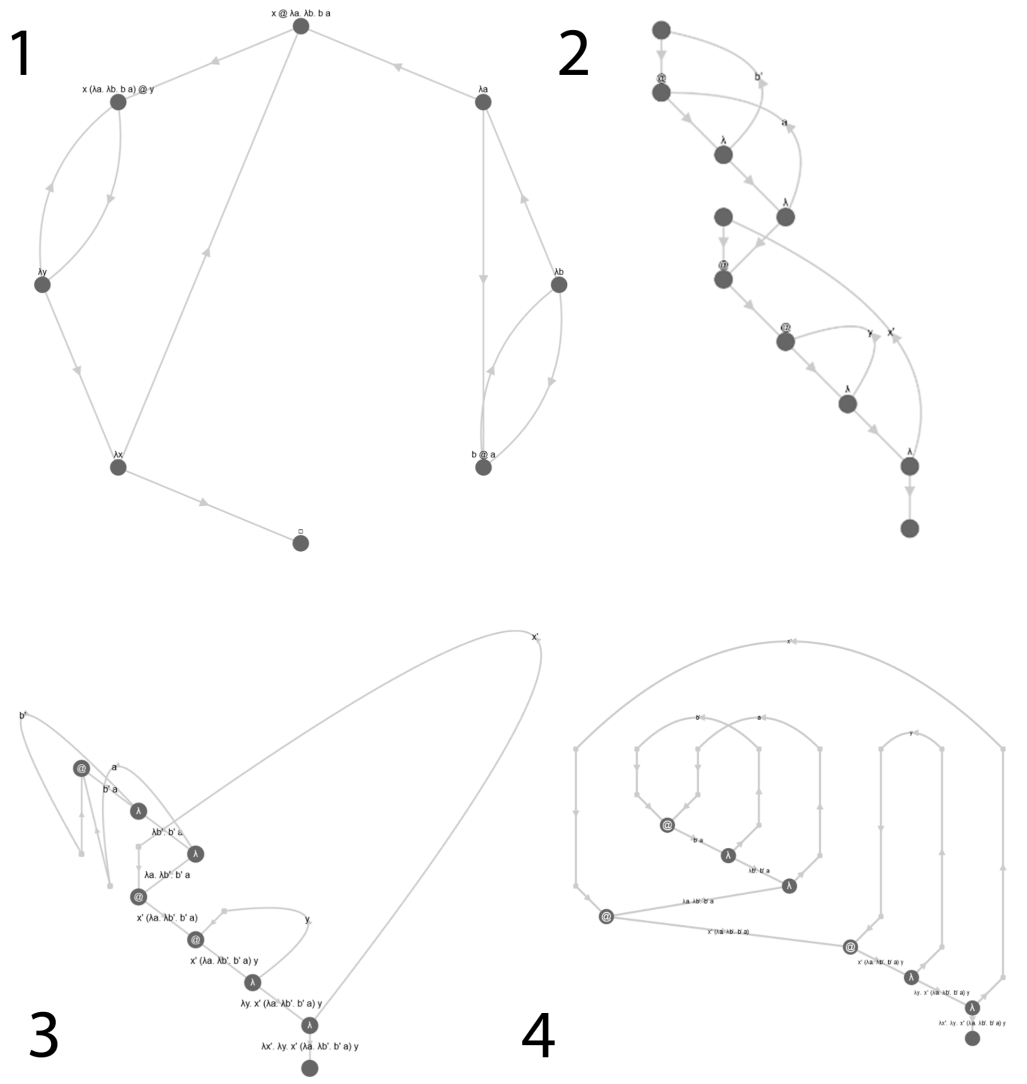


Figure 12: How the map for the term $\lambda x.\lambda y.x\,(\lambda a.\lambda b.b\,a)\,y$ evolved over time. Map 1 is incorrect due to a lack of crossings, Map 2 is incorrect due to too many crossings, Map 3 is not only incorrect due to too many crossings but is also very hard to decipher, Map 4 is the final (correct) version.

node' was added to pull LHS variable edges up in the right direction, away from the RHS edges. The variable edges were changed to use bezier curves, with the intention that these edges would curve around the side of the map and only cross other variable edges when they were supposed to. Unfortunately, the edges still tended to incorrectly cross over other edges due to an insufficiently large curvature. Variables used earlier in the term would not be high up enough on the page to curve around the remainder of the term (seen in Figure 11.2) This most commonly happened with terms containing multiple larger subterms, as variables used earlier in the term would cross over edges leading to the subterms (seen in Figure 12.2 – the only incorrect crossing is created with the edge leading to the $\lambda a.\lambda b.b\,a$ subterm).

After looking into how the bezier curves were drawn, the algorithm was modified slightly for the next version (3). The variable support nodes were placed at the top of the page, with variables used earlier in the term having higher placed support nodes. This was so that the curves would, in theory, avoid the edges created by the rest of the term and only create correct crossings when approaching the abstraction nodes. The support nodes were also given a more subtle style so as not to be confused with the actual nodes of the map. The algorithm appeared to be successful in initial testing (in Figure 11.3), even if it produced slightly ugly maps (e.g. the 'spikes' created by the meeting of the curved edges with the straight ones). However testing with more complex terms (such as in Figure 12.3) revealed huge flaws in the algorithm when dealing with closed subterms. The naive way that earlier variables were placed higher up the page meant that the variables used in the closed subterm were dragged downwards and caused edges to display messily and causing even more incorrect crossings. An oddity in how the curvature of variables edges were rendered can also be seen in Figure 12.3 – the edge representing x has an unnecessarily huge curve, suggesting that the method for calculating the curvature of edges was slightly bugged too.

Since these algorithms were not having much success and the code was growing out of control, it was abandoned and started from scratch (4). Rather than diving straight into creating a new algorithm, I spent some time thinking about a strategy for drawing maps that would work for all terms. Nodes would be drawn in a similar way to before, with the scope of an abstraction heading left of an abstraction node, and the left and right hand side of an application heading out of the appropriate side. This time, however, all of the variable support nodes would be placed at the same height at the very top of the map. Support nodes were also added for abstraction nodes at the same height. This meant that any crossings would only occur at the top of the page, rather than the edges intersecting other parts of the map. The curvature radius was calculated dynamically, based on the distance between the abstraction support node and the variable support node – the further apart they were, the greater the radius. This can be seen in Figure 11.4 – the x edge has the largest radius since it has the furthest to travel. This ensures that all three variable edges do not cross and results in a correct planar map.

Special care was also taken for positioning of subterms. All nodes inside a subterm would be shifted left or right so that they did not intersect with other parts of the map. This is shown in Figure 12.4 – the subterm $\lambda a.\lambda b.b\,a$ has been shifted so it is entirely right of its parent application node, and this application node has also been shifted to the right so that there is enough free space to hold the subterm.

While developing the map-generating algorithm, a recurring problem was labelling edges and nodes correctly. In the API, each element must have a unique `id`, which is used when giving edges a source and target. These ids were added to a λ-context when an abstraction node was encountered so they could be looked up when variables were used later in the term. This caused problems when variable names were used multiple times in a term (such as in $\Omega = (\lambda x.x\,x)(\lambda x.x\,x)$), since uses of the second x would draw edges to the first x. Originally, terms were α-converted during the algorithm (e.g. $\Omega \mapsto (\lambda x.x\,x)(\lambda x'.x'\,x')$), to ensure all variable names were unique. However this meant that the labels on the generated map wouldn't match up with the term specified by the user, potentially making it confusing. Instead, a `label` field was created in map elements to store the original variable name in. This meant that the `id` of each element could still be unique while retaining the original label. Subsequently, when a variable was looked up in the context both the `id` of the abstraction node (for the source of the variable edge) and the label (for labelling the variable edge) would be returned.

Another problem lay in how to deal with free variables. Normally when encountering a variable, it was easy to determine the appropriate abstraction node by prefixing the variable label with a λ. For free variables, this did not work at first because this abstraction node didn't exist! This meant it had to be created during the algorithm, resulting in a clumsy if statement checking for the existence of such a variable and creating a node for it if it didn't exist. This caused all sorts of bugs, such as with labelling as discussed in the previous paragraph. It turned out there was a much simpler solution to this problem – create all the free variable nodes at the very beginning of the algorithm, so they could be treated as ordinary abstraction nodes. Initially these free variables were placed at the top of the page with the rest of the variable edges, but this caused problems when manipulating the maps later as the free and bound abstractions had different named edges. Free variables were changed to use the same basic structure as regular abstractions to ensure consistency for this reason. This provided an important lesson to ensure consistency as much as possible throughout the project – meaning that adding new features later could be done with one block of code rather than have to adjust it for different aspects.

Although representing linear terms was the main goal of creating the visualiser, it turned out that pure terms could also be represented using the same algorithm without needing to make any changes. Because variables and abstractions are considered separately, no problems are caused if variables are used multiple times: multiple edges can connect to one appropriate support node at the top; or not at all: an edge leaves the abstraction node but terminates at the top of the map. This can be seen

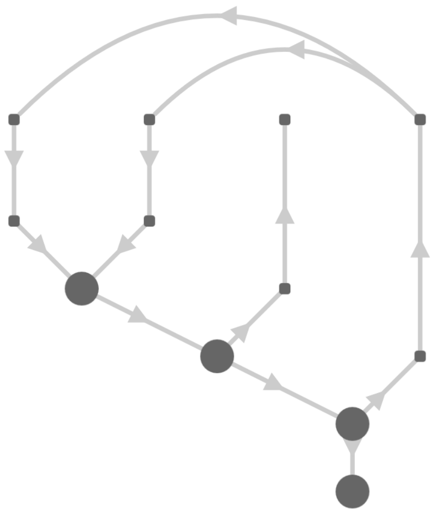


Figure 13: An example of how the map-generating algorithm also works with pure terms, with the term $\lambda x.\lambda y.x\,x$ in which x is used multiple times and y is not used at all.

in Figure 13.

6.5 Counting and generation functions

The next stage of the project was to develop functions to count and generate λ-terms of various fragments of the λ-calculus. These could then be combined with the visualiser to create λ-term galleries.

The number of λ-terms with a given number of subterms n and free variables k can be defined as:

$$\begin{aligned} \mathit{count}(n,k) = {} & \mathit{count}(n-1,k+1) \\ & + \sum_{n_1=1}^{n-2} \mathit{count}(n_1,k)\cdot \mathit{count}(n-1-n_1,k) \\ & + [n=1]\,k \end{aligned}$$

where $[n = 1]\,k$ is equal to k if $n = 1$, 0 otherwise.

The three terms in the sum correspond to abstractions, applications and variables respectively. The number of abstractions can be calculated by counting all terms with one less subterm (the abstraction itself counts for one subterm) and one extra free variable (the abstracted variable). The number of applications is slightly more complicated: we need to account for every possible way of splitting the subterms between the two terms t_1 and t_2. The number of variables is equal to the number of free variables, but only if the number of subterms is equal to 1 – variables can only have one subterm, themselves!

```haskell
data Term = Abs Term | App Term Term | Var Int

gen :: Int -> Int -> [Term]
gen 0 _ = []
gen 1 k = [Var x | x <- [1..k]]
gen n k = [Abs t | t <- gen (n-1) (k+1)]
                ++ [App t1 t2 |
                      n1 <- [1..n-2], t1 <- gen n1 k,
                                       t2 <- gen (n-1-n1) k]
```

Figure 14: A program to generate pure λ-terms of a given number of subterms and free variables.

It is quite simple to develop this equation into a program to generate λ-terms. An example in HASKELL is shown in Figure 14. With some modifications to how we use free variables, we can also create programs to generate planar and linear terms. For planar terms, the context of free variables will be split between the LHS and the RHS of an application, so the algorithm will have to take into account the various points at which it can be split (e.g. for Γ = `[0,1,2]`, the possibilities are `([],[0,1,2])`, `([0],[1,2])`, `([0,1],[2])` and `([0,1,2],[])`). Linear terms are slightly more complex, since the order of the context is not necessarily preserved by the two terms of an application. All the different ways the variables can be split between the LHS and RHS must be considered.

To test that these algorithms were in fact correct, I compared the outputs from the counting algorithms to known sequences on the Online Encyclopedia of Integer Sequences (OEIS). For example, the sequence of numbers of closed linear λ-terms of size n (generated by `[count n 0 | n <- [1..]]` forms the sequence `[0,1,0,0,5,0,0,60,0,0,1105,...]` which corresponds to sequence A062980 on the OEIS (Zeilberger, 2016).

Translating these programs from HASKELL to JAVASCRIPT so they could be used by the tools was fairly simple – pattern matching was replaced by switch statements and list comprehensions by for loops.

6.6 The gallery

With the term generation functions implemented in JAVASCRIPT, the next step was to create 'galleries' to display various sets of terms in. The first thing to consider was the different parameters that could be used to filter terms to create different galleries. The original parameters n and k used in the generating algorithms could be used as a starting point, but there were others too such as number of crossings or number of β-redexes. There may be ways to modify the generation algorithms to only return these terms, but to start with all terms for a given n and k were generated, then the appropriate terms selected from the array. This has the downside of being quite inefficient when it comes to larger arrays of terms but meant that more time could be spent on implementing the tools rather than attempting to devise more algorithms.

Combining the visualiser with the generation algorithms was fairly easy. After receiving user input for values of n and k, the appropriate generation algorithm (pure, linear or planar) would be run to produce an array of λ-terms. These terms could then be fed to the visualiser to produce maps of each of these terms, which would be displayed on screen using basic CSS to arrange them in a grid. Over time the way the gallery was displayed was tinkered with to ensure the best display for all displays and galleries (e.g. terms not too close together, spaced evenly etc.). The ability to select filtering criteria was also added later on – all this required was to use the built-in `filter(x -> ...)` function to remove any terms that didn't fit the chosen criteria. Another modification made to save time when generating galleries of closed terms was to infer an empty k box as a 0 – this saves lots of time when working with closed terms.

A minor problem was found when passing the generated terms to the visualiser. The generated terms did not contain any labels so the visualiser treated every variable as a free variable and didn't connect the edges to the correct abstraction nodes. This was solved by giving abstracted variables unique dummy labels so the visualiser could distinguish between different abstractions.

Originally the captions for the portraits were displayed in de Bruijn notation, to represent the 'structure' of the generated terms rather than associate them with any particular labels. However, this made the captions harder to understand than regular notation. This was fixed by using the labelling function detailed in Section 6.2 to generate a 'canonical' labelling to display alongside the maps. An option to toggle between de Bruijn and regular notation was added, as shown in Figure 15.

As the galleries were tested more it became apparent that for larger galleries, it simply wasn't feasible to display all the maps in a suitable time. Since the actual map-drawing process was the most computationally expensive, an option was added to turn off the map drawing and just show the captions for each term. This meant that a user would still be able to generate the terms, and inspect them in more detail by viewing their 'portrait', as discussed in the next section.

6.7 λ-term 'portraits'

While the galleries were interesting alone to compare the similarities and differences between terms with the same properties, it was hard to inspect the terms in detail from the small portraits. To remedy this, functionality was added so that users could click on a portrait and be shown a larger version, similar to the first visualiser tool.

As in an actual art gallery, I thought it would be interesting to have some of the term's properties displayed next to the large portrait. One of the most interesting

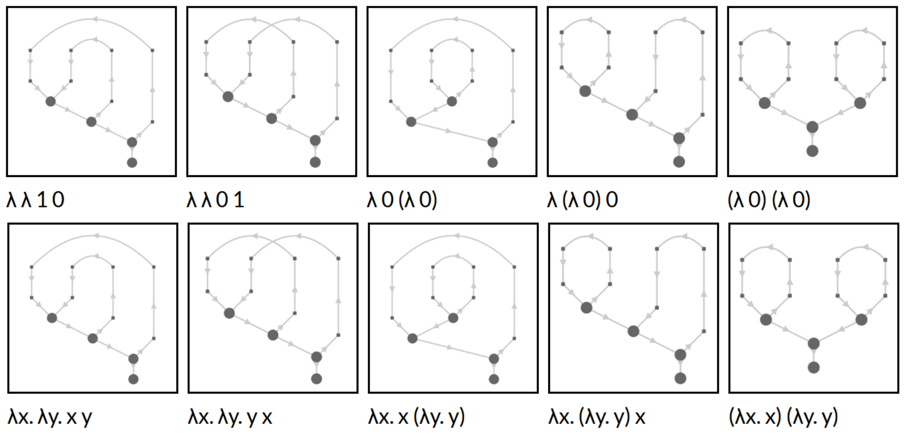


Figure 15: The gallery showing all closed linear terms with 5 subterms, with captions in de Bruijn (top) and their regular 'canonical' (bottom) notation.

topological properties of the maps is crossings, so an algorithm to calculate the number of crossings in a given λ-term had to be implemented. The only edges that cause crossings are those that represent the use of abstracted variables. For a map to have no crossings, its variables must be used in the order they are abstracted, i.e. their de Bruijn indices, when read from left to right, must be in descending order. So to find the crossings created by a particular variable, we can calculate $crossings(x) = |x - (n - 1 - i)|$, where x is the de Bruijn index of the variable, n is the length of the context, and i is the position of the variable in the context. For example, in a list of free variables `[0,1,2]`, the `0` creates two crossings as it must pass through the edges for variables `1` and `2`. Unfortunately, we cannot just perform this calculation for each variable and add them together to find the total number of crossings since this would contain duplicate crossings.

To solve this problem, a slightly different approach was used. This approach was based on the idea that a crossing is created when a free variable on the LHS of an application has a lower de Bruijn index than a variable on the RHS. For example, $0\,(1\,2)$ has two crossings because 0 is less than both variables in the RHS, whereas $2\,(1\,0)$ has no crossings since 2 is greater than both variables in the RHS. These crossings could then be added to the crossings in the LHS and RHS by recursively applying the algorithm. A pseudocode implementation of the calculation of crossings in an application can be seen in Figure 16. It is trivial to consider the other two cases: variables have no crossings and abstractions have the same number of crossings as their scope.

There was initially some trouble translating this into JAVASCRIPT as I made a severe lapse of judgement and tried to incorporate calculating the free variables into the crossings function. Since JAVASCRIPT does not contain tuples, this meant performing various array manipulation options to keep track of the current number of calculated crossings and free variables in the same array – this made the function

```
freeLHS ← freeVariables($LHS$) ;
freeRHS ← freeVariables($RHS$) ;
totalCrossings ← crossings($LHS$) + crossings($RHS$) ;
for $i \leftarrow 0$ to length(freeLHS) do
    for $j \leftarrow 0$ to length(freeRHS) do
        if freeLHS[$i$] < freeRHS[$j$] then
            totalCrossings ← totalCrossings + 1 ;
        end
    end
end
return totalCrossings;
```

Figure 16: Algorithm to calculate crossings in an application.

very hard to understand! I realised that to keep the code cleaner it would be better to first implement a separate function to return an array of free variables in a λ-term. This made the crossings function much tidier and the implementation become very close to the algorithm in Figure 16.

Other facts about the visualised term were much simpler to implement: calculating the number of applications, abstractions and variables involved traversing the term and incrementing a counter whenever a particular element appeared.

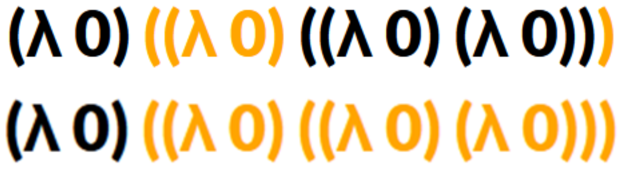
With the λ-term portraits in good condition, I decided to add them to the visualiser as well, to create more consistency between the two tools. This also enabled code to be shared between the tools and caused the overall size of the codebase to decrease, making it easier to maintain.

6.8 Interacting with β-redexes

With the core functionality of the visualiser and the gallery complete, it was time to turn to more specific features. Since one of the main motivations of the project had been to investigate normalisation, β-redexes were a good area to head into next.

I again turned to the ML functions developed by Pierce (2002). Firstly, a function to 'shift' all de Bruijn indices greater than a given number by a certain number of places had to be implemented, followed by a function to substitute a variable for a different term. These could then be combined to create a β-reduction function. At this point, I also made a basic normalisation function that continuously performed an outermost reduction until reaching a normal form or performing a certain number reduction steps. Combining this with the visualiser was again simple: the map drawing function could be called after reducing or normalising the original term to draw the new term.

The next feature to implement was to list the β-redexes alongside the portraits.


(λ 0) ((λ 0) ((λ 0) (λ 0)))

(λ 0) ((λ 0) ((λ 0) (λ 0)))

Figure 17: An example showing how the redex highlighting originally had trouble when dealing with redexes inside redexes, and how it now correctly displays.

This was just a case of traversing the term tree and adding each β-redex to an array as it was encountered, meaning that the leftmost outermost redex would be in position `0` and the rightmost innermost redex would be in position `n`, as shown below:

$$\overbrace{(\lambda x.\, \underbrace{(\lambda y.y)\, x}_{\text{redex 1}})\, (\lambda x.\, \underbrace{(\lambda y.y)\, x}_{\text{redex 2}})}^{\text{redex 0}}$$

These redexes could then be printed next to the portraits. To implement clicking on the redexes to reduce it, a function to perform a specific reduction in a term had to be implemented. This was again a case of traversing the term in the same way with a counter that ticked down until the appropriate redex was encountered.

To allow highlighting of redexes in the original term, a method to print terms with HTML tags had to be implemented. This would print the term with `<span>` tags surrounding each beta redex, so each redex was associated with a class. For example, the term $(\lambda x.x)\, a$ would be enclosed with `<span class="app-0 beta-0">...</span>`. The style for this class could then be changed when highlighting its corresponding redex in the list.

This became slightly more complicated when dealing with redexes within other redexes – the function to change the style of a class had change the style of all classes within the `<span>` tags since the inner classes overwrote the outer ones. This is shown in Figure 17.

A similar approach was used when making the redex highlight on the map. The API used allows elements to have classes, so whenever a redex was encountered a corresponding class would be added to an array, and added to all elements within the redex. Whenever a redex was highlighted, the style function from the API could be called and update all the elements with the appropriate class with the colour.

6.9 Normalisation graphs

One of the main ideas from the project proposal was the use of the visualised maps in normalisation graphs. The first step in this was to implement these normalisation graphs as a data structure in Javascript, and then using these structures alongside the graph drawing API to generate the graphs.

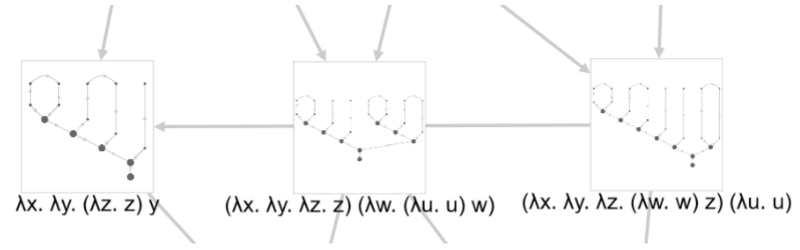

λx. λy. (λz. z) y
(λx. λy. λz. z) (λw. (λu. u) w)
(λx. λy. λz. (λw. w) z) (λu. u)

Figure 18: An example of when the generated normalisation graph can feature edges than cross behind other nodes for pure terms (the edge from the right node to the left node crosses behind the centre node). However it is a lot harder than expected to find examples like this.

Originally normalisation graphs were created as a naive tree structure. The root node contained the original term, with children representing all possible reductions of that term (each edge representing an individual redex). While this seemed like a suitable idea in theory, it was flawed in that not all redexes lead to unique reductions (e.g. the two redexes in $(\lambda x.(\lambda y.(\lambda z.z)\,y)\,x)$ lead to α-equivalent terms), but the tree structure could not represent two edges leading to the same node. This also meant that the normalisation graph would not converge to the normal form at the bottom, but would diverge to many nodes containing the same normal form. This implementation would also fail to detect normalisation loops and would continue to descend the tree infinitely. However it had the benefit of being simple to implement and understand.

Regardless of its problems, this implementation was used for the first iteration of generating normalisation graphs. Since the graph API had the ability to create 'compound nodes' (nodes with other graph elements inside them), the first strategy was to traverse the normalisation tree, generating maps of the terms at each node and placing them inside the larger node of the normalisation graph. The ids of elements would have to be renamed so that maps from different nodes didn't link up by mistake, but this was just a case of suffixing a unique id after each map's elements. These normalisation graph nodes would then be connected by the edges representing redexes. To place the nodes, each reduction was given a 'level' value which specified how many steps had been taken to reach it from the original term. Nodes with the same level were spread out evenly in a row, with rows starting from the top of the page and moving down. This works fine for linear terms, since steps always take you further down the normalisation graph due to all paths being the same length (it is impossible to get to another reduction on the same level). However for pure terms this can cause some irregularities since consecutive reductions can occupy the same level. Since edges are drawn in straight lines, this means they can cut across other nodes in the graph, as shown in Figure 18. Fortunately this doesn't happen too frequently, and the nodes can be moved out of the way of the edge if desired.

The first attempt at drawing normalisation graphs had numerous problems. While

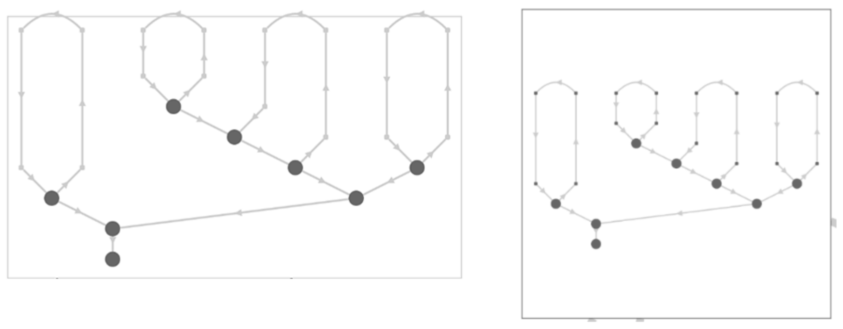


Figure 19: A comparison between the compound node (left) and background image (right) versions of normalisation graph nodes.

the API didn't redraw nodes with duplicate ids (which I thought might remove the problem of the duplicate reductions in the normalisation tree), there was a problem with edges – each node of the tree would attempt to draw its own set of edges, causing many duplicates. This was fixed by checking for unique reductions in the graph drawing algorithm, but this was unwieldy and made the algorithm quite hard to understand. Compound nodes also caused the performance of the map to drop drastically – moving nodes around was especially laggy. This was not helped by each duplicate node drawing its own map, which I initially didn't realise was happening since the maps were drawn on top of each other. Finally there were no options to give a margin around the maps in the normalisation graph nodes, so some of the edges would intersect with the edge of their parent node, as shown in the left of Figure 19.

In an attempt to reduce the lag created by the drawn graph, I switched to using images for the normalisation graph nodes. The map of the reduction would be generated first, then converted to an image using the API functions. This image could then be used as the background of the normalisation graph node, as shown in the right of Figure 19. This made creating tidier nodes much easier as a margin could be specified around the image so everything fit inside. Options were also added to disable map generation for large terms (like with the gallery) to reduce the generation time of the graph. It also turned out that drawing arrows on the edges was a large cause of lag, so an option to not draw these was added.

While the tree implementation of normalisation graphs had just about been holding up through all of this tweaking, it started to show severe weaknesses when attempting to implement functions to calculate path length. Because of the numerous duplicate nodes, complex methods needed to be implemented to ensure that only unique paths were being calculated. This was quite complicated and bloated the functions significantly. I decided that it was time to migrate to a more sophisticated implementation.

Rather than a tree, the new implementation uses an adjacency matrix that keeps track of which nodes connect to each other, and by which redex. The original term is fed to the initialisation algorithm, which then performs all the redexes in the term and adds the resulting reductions to a frontier. The current node is added to the adjacency matrix, with references to the nodes it connects to. This process is continued, with any previously seen reductions not being re-added to the frontier, until the frontier is empty and the graph is complete. This has several benefits over the tree implementation. Most notably, every node is unique, which also means that there are no duplicate edges that need to be removed when creating the map. Additionally, because redexes are implemented as references to already existing objects rather than new child objects, infinitely reducing terms with finite normalisation graphs (such as $(\lambda x.x\,x)(\lambda x.x\,x)$) can have these graphs displayed. The only time the algorithm could carry on infinitely would be if the term reduces infinitely with an infinite normalisation graph, such as for $(\lambda x.x\,x\,x)(\lambda x.x\,x\,x)$. For these terms, a cutoff point was added to halt the algorithm at a suitable number of reductions. In the code, the algorithm becomes much more elegant (both while generating the graph and drawing it), since recursion into the tree is replaced with a loop across the adjacency matrix. The value for 'level' was calculated in a similar way to before, with all reductions from the original term assigned level 1, and each subsequent redex increasing this value by 1.

Path functions were now easier to implement. The total number of paths and list of different path lengths could be calculated by starting at the original term and following the references in the adjacency matrix until the normal form was reached. Properties such as shortest or longest path could then be computed by performing operations on the list of path lengths, such as `min()` or `max()`. For larger normalisation graphs it turned out that it took a very long time to generate all the different path lengths across thousands of edges, so these functions were made an optional extra to view rather than being generated for all graphs by default.

A feature added slightly later on was the ability to hover over edges and see the redex that that it represented. The API provided functions for performing actions on mouse events, so this was easy to implement. The only minor problem was that redexes could free variables that were bound in the original term, so when it came to printing them there was no way of knowing which variable they referred to. For example, $\lambda\,(\lambda\,0)\,0$ prints as $\lambda x.(\lambda y.y)\,x$ but the redex $(\lambda\,0)\,0$ on its own prints as $(\lambda y.y)\,?$, since the x is never bound. This was solved by storing the correctly printed version of the redex at graph initialisation (by considering the original term) rather than trying to generate the label at run-time.

6.10 Animating normalisation

The last major feature implemented was animating the normalisation of terms. In theory, this was simple: highlight the appropriate redex, then perform it and redraw the new term, then keep going until the normal form was reached. Unfortunately, chaining these operations was slightly harder than anticipated since JAVASCRIPT is single-threaded and as such doesn't have a conventional `sleep()` function like in multi-threaded languages. Instead it has a `setTimeout()` function that is passed a function and executes it after a given time. This meant that what was originally a simple while loop had to become a recursive function with several nested `setTimeout()` functions for the process of highlighting and performing each redex, then calling the animation function on the new term.

Three strategies were implemented: leftmost outermost, which performs the first redex in the redex list, rightmost innermost, which performs the last redex in the redex list, and random.

6.11 Finishing touches

With all main functionality completed, all that was left to do was to polish up the interface. To make use of the tool easier, the page would automatically scroll down to the appropriate section of the page once buttons were clicked, rather than making users scroll down themselves. I also tried to make the tools display better on different displays by using relative rather than absolute values for element dimensions in the CSS file. This meant that the visualised map would fit to the screen size rather than spilling over, as had been happening on smaller screens.

One thing I noticed was how large maps and graphs were quite hard to view, so I implemented a 'full screen' mode to redraw them to fill the screen. This was a simple case of changing the dimensions of the element the map was drawn in to fill the entire screen, then redrawing the map inside it. Unfortunately this can cause delays for large terms since the entire map generating algorithm has to be run again, but since this was being done at the end of the project there wasn't enough time to look into making an algorithm to re-position all the elements one by one.

7 Features and Examples

This section details the features that are implemented in the final version of the tools, and examples of ways they could be used.

7.1 Drawing λ-terms

The core feature of the tools is the ability to represent any λ-term from user input as a rooted map on the screen. As can be seen in Figure 20, the visualiser is not limited to linear terms – maps for any pure λ-term can be generated.

A context of free variables can also be specified, allowing these variables to be used in the visualised term. Terms using the free variables a and b can be seen at the bottom of Figure 20, showing how the free variables are placed to the right of the map. The user can pan and zoom around the map to inspect it more closely. Nodes can also be dragged around to new positions. The user can also choose to display the visualisation with or without labels (if just the structure is of interest). 'Full screen' mode is available so the visualised terms can fill the screen.

To make it easier to input large expressions, users can associate them with **aliases** (e.g. `id` $= \lambda x.x$) to reduce the amount of typing necessary. An example using the Church encoding of pairs (`Pair` $= \lambda x.\lambda y.\lambda v.\, v\, x\, y$ and `fst` $= \lambda p.\, p\, (\lambda x.\lambda y.\, x)$) is shown in Figure 21. A large number of aliases can be pasted in at once with the 'bulk alias' function, which can be useful when using a large amount of predefined functions (such as Mairson's encodings of boolean circuits, as shown in Section 7.3).

A number of properties are shown alongside the visualised term. Of most interest are the number of crossings, which might be hard to count if there are many, and number of β-redexes, which are not always obvious (and could be hidden inside aliases).

7.2 Normalisation

The β-redexes in the term are listed next to the term when it is visualised, as shown in Figure 22. By hovering over them, a user can see them highlighted in the visualisation and within the term itself. Different redexes are highlighted in different colours to distinguish between them clearly, as shown in Figure 23. By clicking a redex, it will be performed and the visualisation updated with the newly reduced term. By continuing to click on redexes the normal form (if it exists) can be eventually reached. Alternatively, by clicking the 'normalise' button the term will display its normal form straight away if it exists. For example, the term `fst (Pair x y)` from Figure 21 normalises to x, as expected.

Rather than clicking on the redexes, the process can be animated. There are three strategies provided: left-to-right outermost, right-to-left innermost, and random. The visualiser will highlight the redex it is about to perform, then perform it. This process

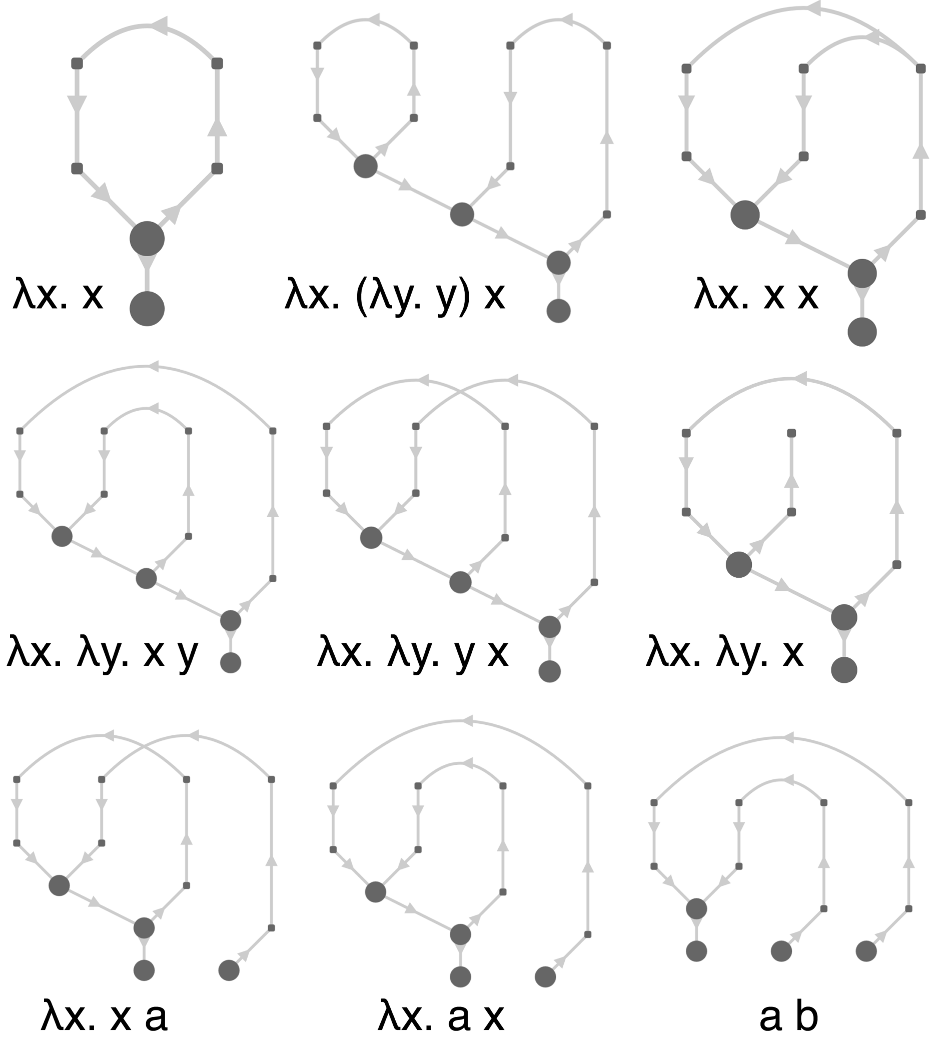
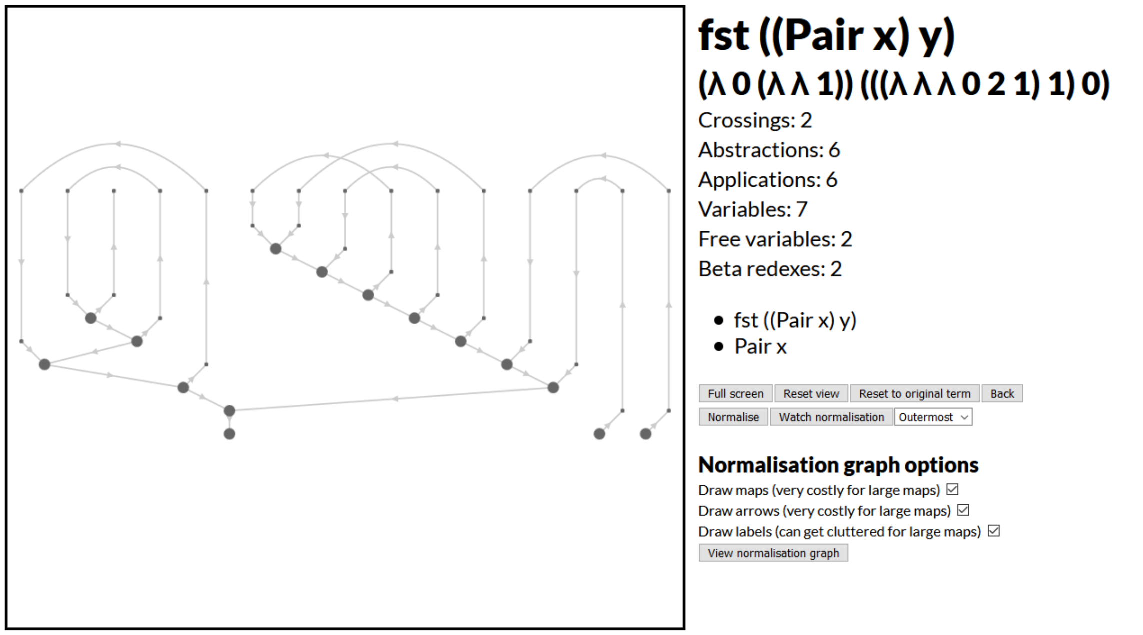


Figure 20: A selection of terms as generated by the visualiser, where a and b are free variables. It is easy to see which terms are linear, as the nodes are all 3-valent, and which are planar, since there are no crossings.



fst ((Pair x) y)

(λ 0 (λ λ 1)) (((λ λ λ 0 2 1) 1) 0)

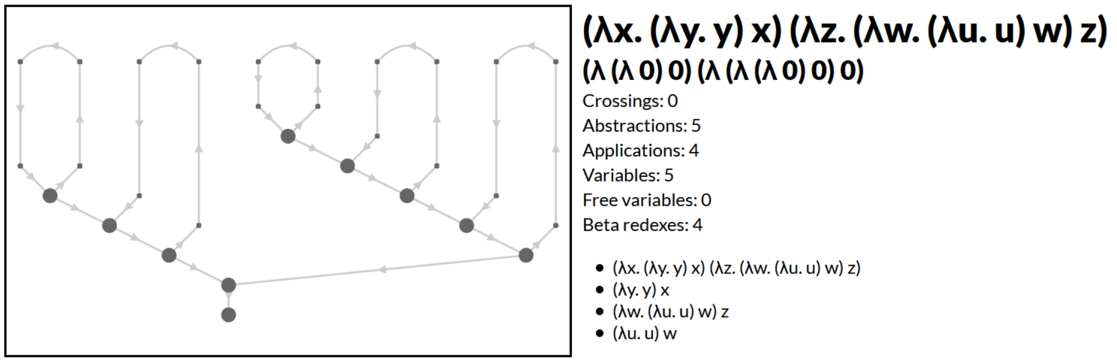
Crossings: 2
Abstractions: 6
Applications: 6
Variables: 7
Free variables: 2
Beta redexes: 2

- fst ((Pair x) y)
- Pair x

Normalisation graph options

Draw maps (very costly for large maps) ☑
Draw arrows (very costly for large maps) ☑
Draw labels (can get cluttered for large maps) ☑

Figure 21: The function `fst (Pair x y)` represented in the visualiser using aliases.



(λx. (λy. y) x) (λz. (λw. (λu. u) w) z)

(λ (λ 0) 0) (λ (λ (λ 0) 0) 0)

Crossings: 0
Abstractions: 5
Applications: 4
Variables: 5
Free variables: 0
Beta redexes: 4

- (λx. (λy. y) x) (λz. (λw. (λu. u) w) z)
- (λy. y) x
- (λw. (λu. u) w) z
- (λu. u) w

Figure 22: The visualiser output for the term $(\lambda x.(\lambda y.y)\,x)(\lambda z.(\lambda w.(\lambda u.u)\,w)\,z)$, listing all four β-redexes.

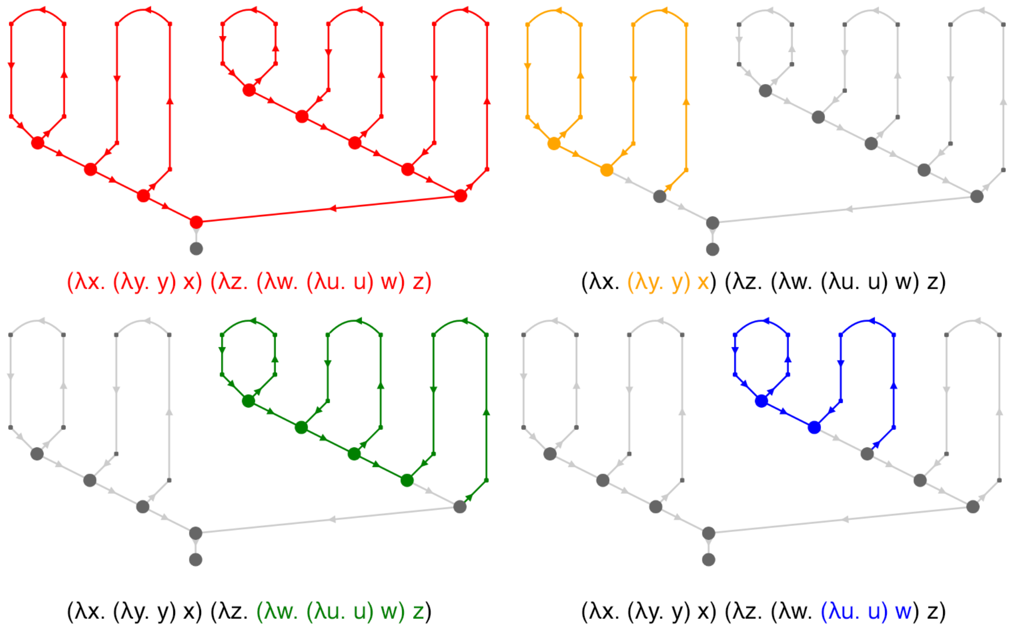

(λx. (λy. y) x) (λz. (λw. (λu. u) w) z)
(λx. (λy. y) x) (λz. (λw. (λu. u) w) z)
(λx. (λy. y) x) (λz. (λw. (λu. u) w) z)
(λx. (λy. y) x) (λz. (λw. (λu. u) w) z)

Figure 23: The four redexes in the term $(\lambda x.(\lambda y.y)\,x)(\lambda z.(\lambda w.(\lambda u.u)\,w)\,z)$, highlighted in the visualiser.

repeats until the normal form is reached (or might just continue forever if there is no computable normal form). An example of this process is shown in Figure 24.

A normalisation graph can also be generated, as shown in Figure 25. By default, this shows an image of the corresponding map at each reduction node, however this can be disabled and replaced with a blank node for large graphs since it can take a long time to render. Path lengths (shortest, longest, average etc) can also be generated. From these normalisation graphs we can observe some of the special normalisation properties that linear terms have. For example, the graph for the planar term in Figure 25 has several different paths to its normal form, but all these paths have the same length. It is also clear to see from the generated maps for each node that planarity is preserved throughout normalisation.

Some terms, such as Ω, have no computable normal form, but do have finite normalisation graphs. These can also be generated by the visualiser. An example that combines Ω with a path that does reach a normal form is shown in Figure 26. For terms that do have infinite normalisation graphs, there is a cutoff point at which the normalisation process terminates and the start of the normalisation graph is displayed.

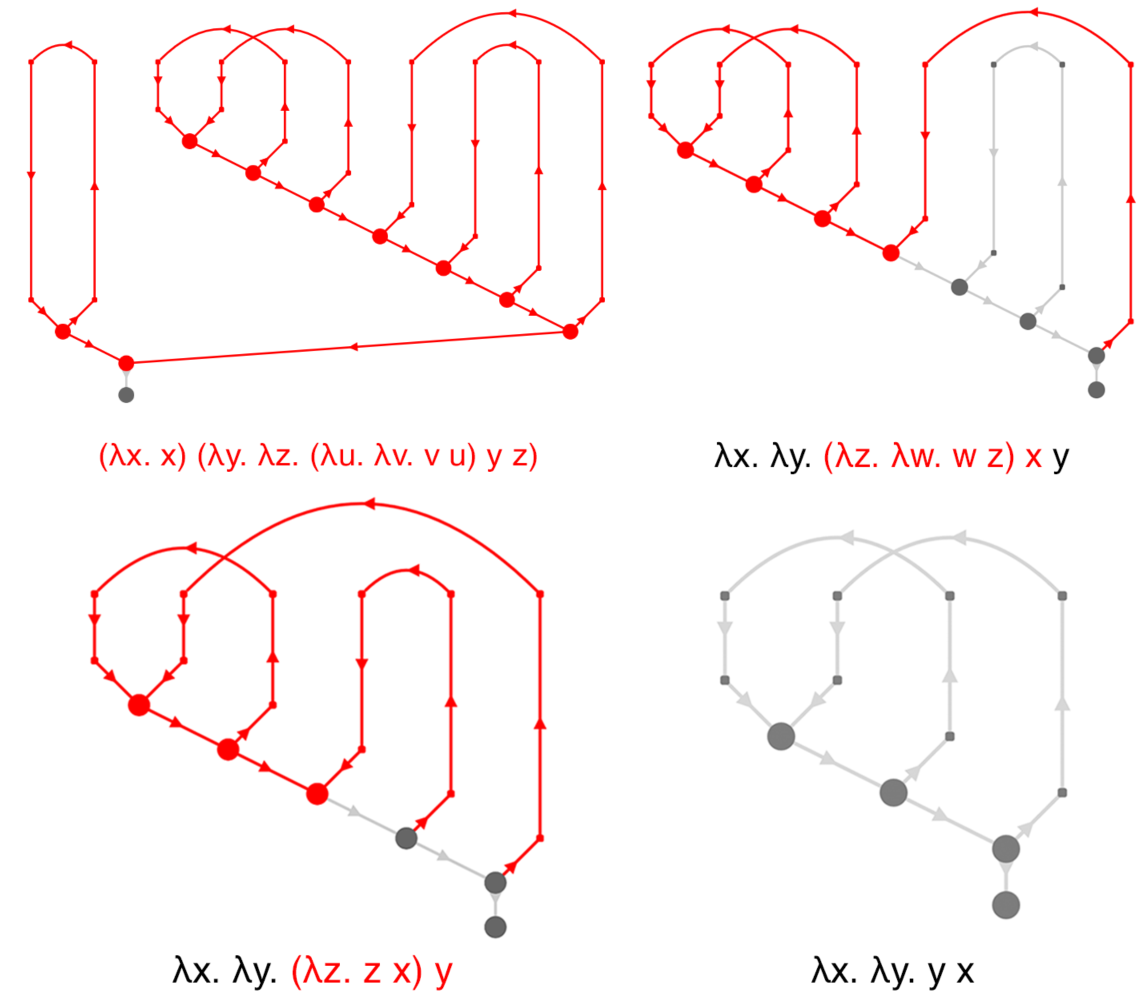

(λx. x) (λy. λz. (λu. λv. v u) y z)
λx. λy. (λz. λw. w z) x y
λx. λy. (λz. z x) y
λx. λy. y x

Figure 24: The steps taken by the visualiser to perform a leftmost outermost normalisation of $(\lambda x.x)(\lambda y.\lambda z.(\lambda u.\lambda v.v\, u)\, y\, z)$. The red highlighting represents the redex about to be performed.

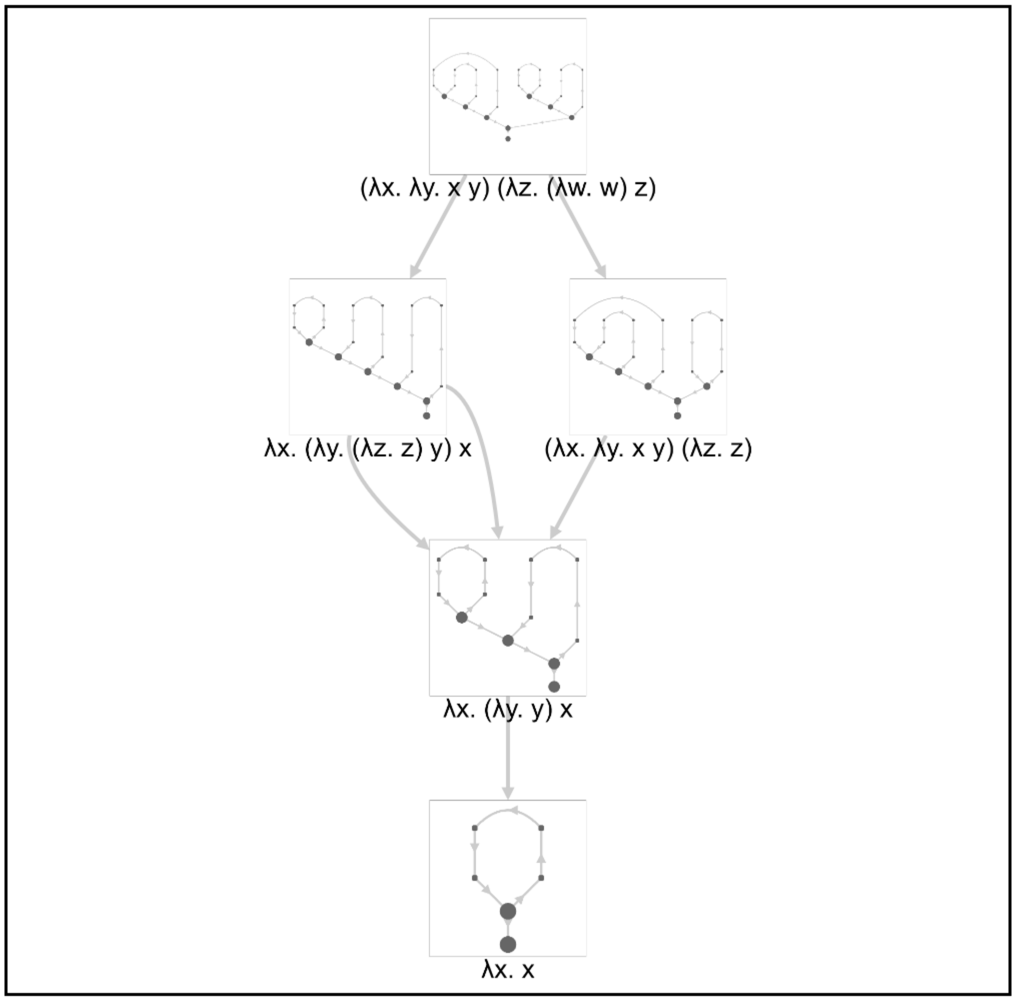


Figure 25: The normalisation graph for $(\lambda x.\lambda y.x\,y)(\lambda z.(\lambda w.w)\,z)$, showing the diverging paths that eventually converge to the normal form.

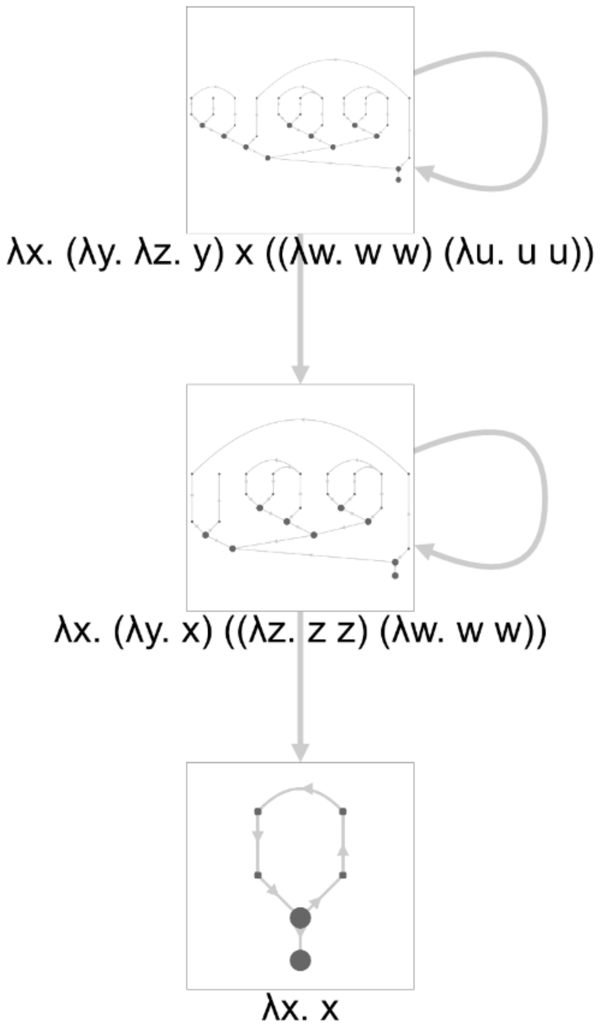

λx. (λy. λz. y) x ((λw. w w) (λu. u u))
λx. (λy. x) ((λz. z z) (λw. w w))
λx. x

Figure 26: Normalisation graphs can contain loops, such as this one for the term $\lambda x.(\lambda y.\lambda z.y)\, x\, ((\lambda w.w\, w)(\lambda u.u\, u))$.

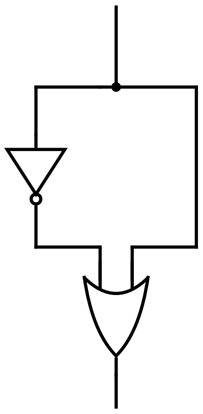


Figure 27: A simple boolean circuit, representing `x -> Or (Not x) x`.

7.3 Example: Boolean circuits

To prove that normalisation of linear λ-terms is PTIME-complete, Mairson (2004) encoded boolean circuits as functions in ML. By normalising these functions, the output of the boolean circuit can be produced. Since the Circuit Value Problem is known to be PTIME-complete, reducing it to the normalisation of linear λ-terms shows that the latter is also PTIME-complete.

It can be interesting to use the visualiser to view the structure of the maps associated with these. One example is the circuit shown in Figure 27, which is encoded as $\lambda x.\ \texttt{Copy}\ x\ (\lambda x_1.\lambda x_2.\ \texttt{Notgate}\ x_1\ (\lambda x_3.\ \texttt{Orgate}\ x_3\ x_2\ (\lambda x_4.x_4)))$. The `Copy` function represents a fanout gate, copying one input x to two outputs, x_1 and x_2. The `Notgate` function takes the input x_1 and negates it to to the output x_3. The `Orgate` function takes in two inputs x_2 and x_3 and returns the `Or` of these as x_4.

The corresponding map is shown in Figure 28, demonstrating how very simple circuits can create quite large maps! If we give the circuit an input, we can use the visualiser to step through the redexes and arrive at the normal form, which will be the output of the circuit. The circuit in Figure 27 applied to `True` normalises to the pair representing `True`, as expected.

In theory, the normalisation graphs for these circuits could also be generated to view all the different paths normalisation could take. Unfortunately, this appears to be infeasible on desktop machines due to the huge number of nodes in the graph. However, the normalisation graphs for smaller terms *can* be generated in a suitable time, such as the one representing `And True False` in Figure 29. By zooming in using the visualiser, the structure of this graph can be inspected.

7.4 Generating λ-term galleries

Galleries of λ-terms with a given number of subterms and free variables can be generated. These galleries display all the maps for these terms in a grid. Captions can be

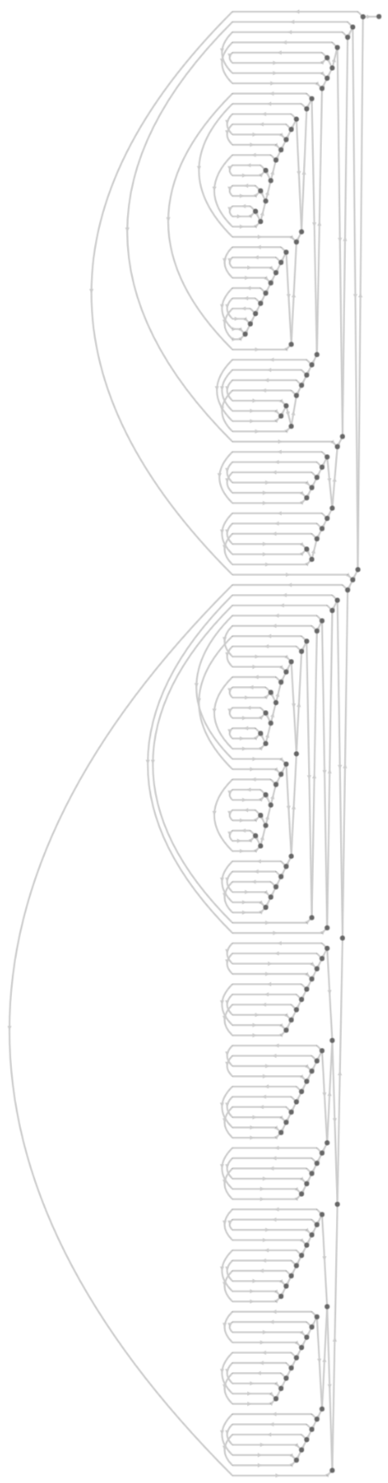


Figure 28: The corresponding map for the circuit in Figure 27

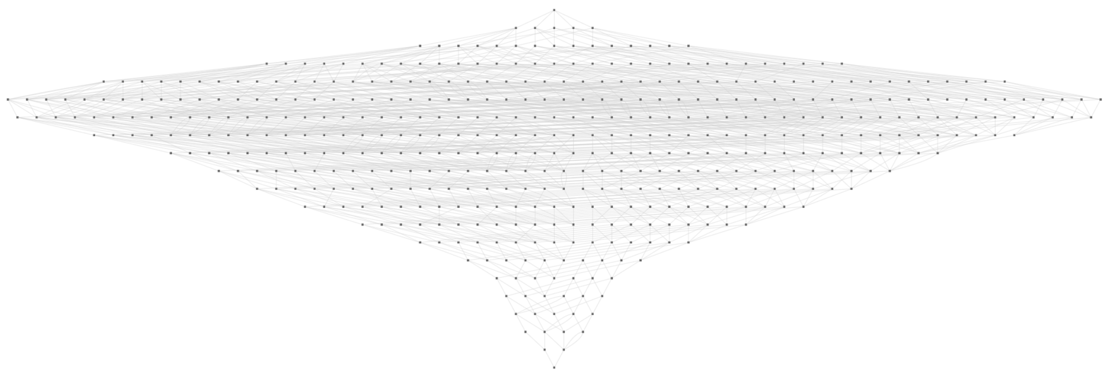


Figure 29: The structure of the normalisation graph for `And True False`.

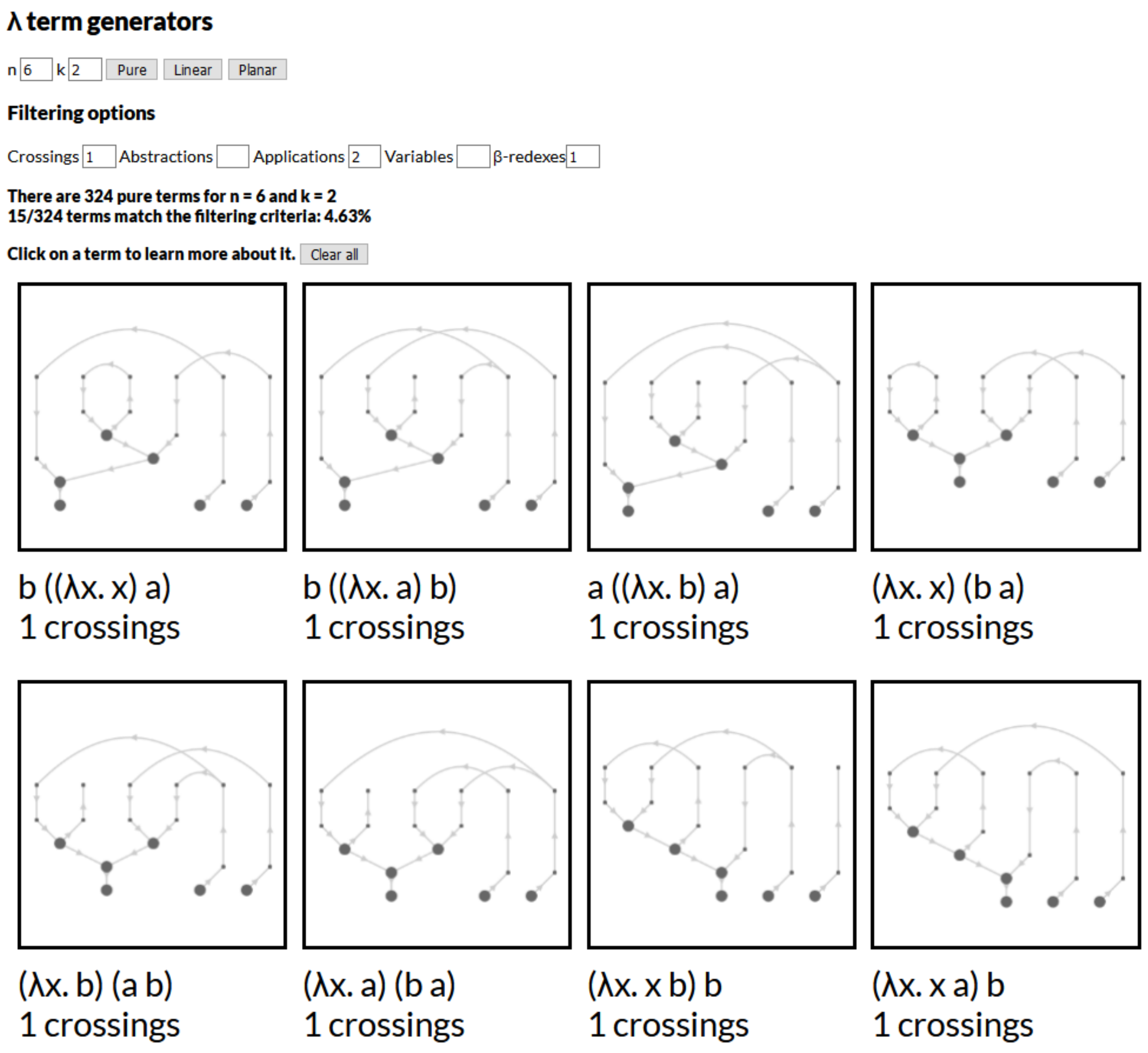
shown in regular or de Bruijn notation. For larger galleries, the maps can be turned off to save time when generating them. The results can be filtered by a number of properties, such as crossings and β-redexes. The number of these terms and the proportion from the full set of terms is displayed. An example is shown in Figure 30.

By clicking on one of the 'portraits' in the gallery, the user will shown what what they would have seen had they typed that term into the visualiser.

7.5 Example: Testing conjectures

When researching λ-terms and their maps, many conjectures can be made. Ordinarily, testing these conjectures would require drawing out the various maps by hand to check that no counterexamples exist. For larger fragments of the λ-calculus this could take a very long time! By using the λ-term generator, this problem is eliminated and counterexamples can be found very quickly.

One very simple (and obviously false) example of a conjecture could be *All closed linear terms of size 11 are planar.* There are 1105 linear terms of size 11, so drawing out all of these would be infeasible by hand. Instead, we can use the gallery, with the crossings criteria set to 0. Since the gallery only generates 336 out of the 1105 terms (as shown in Figure 31), we now know that there are 769 counterexamples to our conjecture, disproving it. We could then adjust our conjecture to be *All closed linear terms of size 11 with 3 β-redexes are planar.* The gallery generates 14 terms that satisfy this criteria and, indeed, they are all planar, as shown in Figure 32. One thing I noticed while testing some of these example conjectures was that closed linear terms of size 8 with two β-redexes and size 5 with one β-redex are also all planar, also shown in Figure 32. This could lead to the conjecture *All closed linear terms of size n with $\frac{n-2}{3}$ β-redexes are planar.* This also holds for size 14 with 4 β-redexes, but cannot be tested much further since my computer cannot generate larger terms in a suitable time. Nevertheless, since we have found no counterexamples there is a strong basis for the conjecture holding for larger values of n.



λ term generators

n 6 k 2 Pure Linear Planar

Filtering options

Crossings 1 Abstractions Applications 2 Variables β-redexes 1

There are 324 pure terms for n = 6 and k = 2
15/324 terms match the filtering criteria: 4.63%

Click on a term to learn more about it. Clear all

b ((λx. x) a)
1 crossings

b ((λx. a) b)
1 crossings

a ((λx. b) a)
1 crossings

(λx. x) (b a)
1 crossings

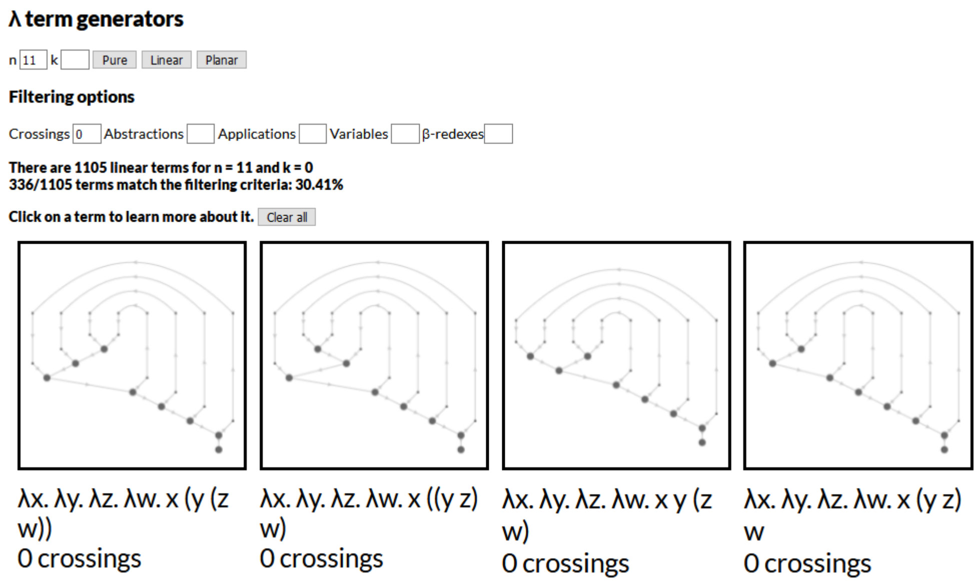
(λx. b) (a b)
1 crossings

(λx. a) (b a)
1 crossings

(λx. x b) b
1 crossings

(λx. x a) b
1 crossings

Figure 30: The beginning of the gallery containing the 15 pure terms of size 6 with 2 free variables, 1 crossing, 2 applications and 1 β-redex



λ term generators

n 11 k Pure Linear Planar

Filtering options

Crossings 0 Abstractions Applications Variables β-redexes

There are 1105 linear terms for n = 11 and k = 0
336/1105 terms match the filtering criteria: 30.41%

Click on a term to learn more about it. Clear all

λx. λy. λz. λw. x (y (z w))
0 crossings

λx. λy. λz. λw. x ((y z) w)
0 crossings

λx. λy. λz. λw. x y (z w)
0 crossings

λx. λy. λz. λw. x (y z) w
0 crossings

Figure 31: Testing the conjecture *All closed linear terms of size 11 are planar* – only 30.41% of terms of size 11 satisfy this criteria so the rest are all counterexamples, disproving the conjecture.

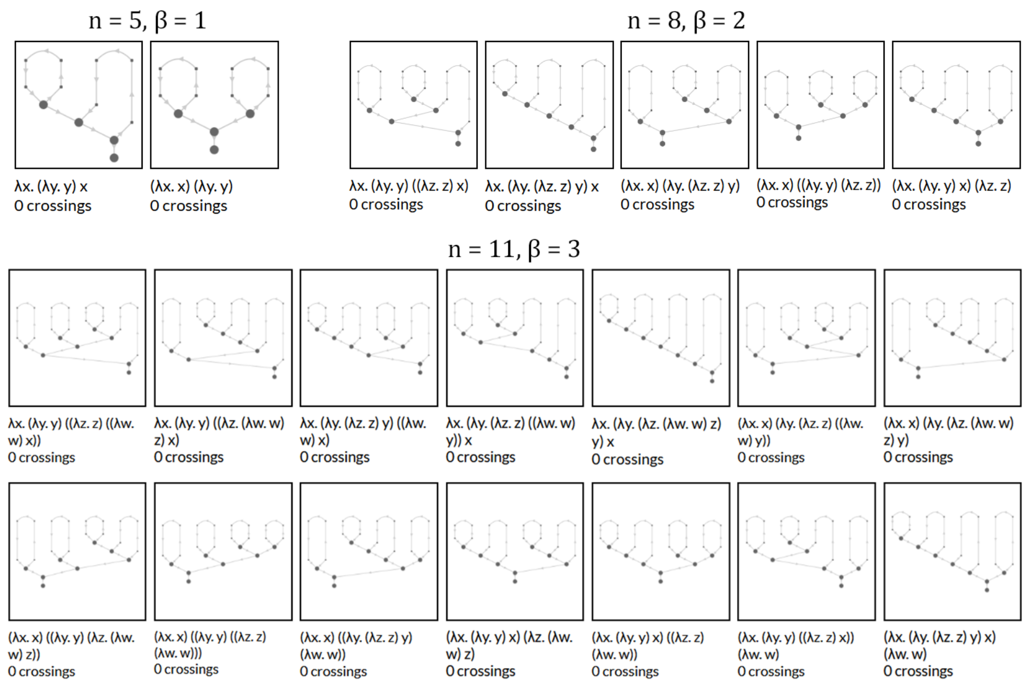

n = 5, β = 1
λx. (λy. y) x
0 crossings
(λx. x) (λy. y)
0 crossings
n = 8, β = 2
λx. (λy. y) ((λz. z) x)
0 crossings
λx. (λy. (λz. z) y) x
0 crossings
(λx. x) (λy. (λz. z) y)
0 crossings
(λx. x) ((λy. y) (λz. z))
0 crossings
(λx. (λy. y) x) (λz. z)
0 crossings
n = 11, β = 3
λx. (λy. y) ((λz. z) ((λw. w) x))
0 crossings
λx. (λy. y) ((λz. (λw. w) z) x)
0 crossings
λx. (λy. (λz. z) y) ((λw. w) x)
0 crossings
λx. (λy. (λz. z) ((λw. w) y)) x
0 crossings
λx. (λy. (λz. (λw. w) z) y) x
0 crossings
(λx. x) (λy. (λz. z) ((λw. w) y))
0 crossings
(λx. x) (λy. (λz. (λw. w) z) y)
0 crossings
(λx. x) ((λy. y) (λz. (λw. w) z))
0 crossings
(λx. x) ((λy. y) ((λz. z) (λw. w)))
0 crossings
(λx. x) ((λy. (λz. z) y) (λw. w))
0 crossings
(λx. (λy. y) x) (λz. (λw. w) z)
0 crossings
(λx. (λy. y) x) ((λz. z) (λw. w))
0 crossings
(λx. (λy. y) ((λz. z) x)) (λw. w)
0 crossings
(λx. (λy. (λz. z) y) x) (λw. w)
0 crossings

Figure 32: Testing the conjecture *All closed linear terms of size* n *with* $\frac{n-2}{3}$ β*-redexes are planar.*

8 Testing

After the development of each major feature in the tools, unit testing was used to check whether the criteria specified in Section 5 were satisfied. The results of the tests could then be used to locate any bugs remaining in the code. A selection of the tests can be found in the `tests` directory in the project repository.

8.1 Parsing terms from user input

Correctness Checked by entering a variety of different valid terms and ensuring that the de Bruijn notation generated by the program matched the version computed by hand. These tests all **passed**.

Reliability Checked by entering a variety of different invalid terms and ensuring that they were all rejected. These tests all **passed**.

8.2 Visualising terms

Correctness Checked by comparing the visualised maps for a variety of different terms with hand-drawn versions of these maps, and ensuring the adjacency relations and order of edges around nodes were preserved. Some larger terms were also generated and the maps checked by eye to ensure they were correct. These tests all **passed**.

Clarity Checked by making sure all maps generated were easy to understand. These tests all **passed**.

Consistency Checked by making sure similar terms produced similarly structured maps. These tests all **passed**.

Performance Checked by making sure maps were generated quickly. Larger terms can take longer to generate, so it is likely that loading times might become too long for huge terms, but for normal use these tests **passed**. However manipulating larger terms can become quite laggy (a problem with the graph drawing API), which is a problem to keep in mind.

Completeness Checked by making sure that all tests passed for a variety of different terms from different fragments of the λ-calculus. These tests all **passed**.

Aesthetics Checked by making sure all the generated maps were pretty. This is of course heavily subjective, but by showing several people the consensus was that these tests **passed** – personally I find viewing them extremely satisfying.

8.3 Generating λ-terms from a given fragment

Completeness The complete list of terms from some smaller fragments were written out and compared with the terms displayed in the gallery, making sure that all of the written terms were present. These tests all **passed**.

Soundness Some criteria were specified for the galleries generated above, and the terms in the gallery checked to make sure they all satisfied the criteria. These tests all **passed**.

Clarity This was checked by eye by generating arbitrary galleries and ensuring they displayed tidily in a grid each time. These tests all **passed**.

Performance This was checked by attempting to generate lots of different sized galleries and checking if they loaded within a suitable time (less than 5 seconds). While smaller galleries normally didn't take more than a few seconds to load, larger galleries (e.g. with $n > 10$) had a noticeable delay. While disabling map drawing would help with this, the larger galleries still did not load in a suitable time. Some tests therefore **failed**.

8.4 Investigating normalisation properties of fragments

Correctness For the listed β-redexes each one was performed and checked by hand to be correct. For graphs, this was checked by drawing some normalisation graphs by hand and comparing them to the generated graphs. These tests all **passed**.

Completeness For β-redexes in terms, this was checked by inputting lots of different terms and checking that all β-redexes were displayed. For graphs, this was checked by generating many different normalisation graphs (including ones with loops) and checking that they were all valid. Complete graphs for terms with infinite reduction graphs was of course impossible, but they were checked up to the visualiser's cutoff point. These tests all **passed**.

Performance This was checked by attempting to generate some large normalisation graphs (such as ones for the boolean circuits) and making sure they generated within a suitable time (less than 10 seconds). Unfortunately, like with the gallery generation, this was shown to be impossible for many graphs and they would never generate in time. So some tests **failed**.

9 Evaluation

9.1 Achievements

From the testing carried out in Section 8, it is clear to see that the tools are quite successful in satisfying the criteria set out in the Specification. There are many different functions provided by the tool, which can all help in research and experimental mathematics involving the λ-calculus and drawing λ-terms as rooted maps:

- The visualiser can generate the corresponding maps for any λ-term from user input. This enables the structure of the maps to be examined without having to draw them by hand.
- The visualiser can display properties of the generated maps, such as crossings. This means that crossings do not have to be counted manually, which could be tricky on large maps, and boring if crossings from many different maps had to be counted.
- The visualised maps can be reduced by choosing redexes inside them. This allows us to view how the map changes on a particular path to a normal form (and other interesting things, such as linearity being preserved for linear terms).
- Normalisation can be animated, showing us how different reduction strategies impact the structure of the corresponding maps.
- Normalisation graphs can be generated for terms with and without normal forms, providing this graph is finite. A small portion of this graph is displayed if the graph is infinite. This enables closer study of these normalisation graphs without having to draw them by hand. It also gives us a different way to study how the maps change in different ways on different paths to a normal form. Viewing these graphs might give some insight into more normalisation properties of linear and planar terms.
- The gallery can generate many different galleries containing terms of a certain size and number of free variables, and filter them by more criteria, such as β-redexes or crossings. This allows us to quickly see all the terms that fulfil some criteria, meaning we can find counterexamples to conjectures, or attempt to find sequences if we are interested in counting these terms.

9.2 Performance issues

However, the testing also highlighted a key problem with the tools – issues with their performance. There are two major areas where poor performance is noticeable.

9.2.1 Lag experienced with larger terms

As terms grow in size, noticeable lag begins to occur when interacting with the maps in the visualiser, or hovering over the listed β-redexes. This seems to be a problem with the graph drawing API – from looking at the documentation it seems that drawing edge arrows might be the main cause of lag. Arrows are not essential for the maps to be correct, but they are useful additions – an option could be added to remove arrows if the problem becomes particularly noticeable.

It is possible that more efficient APIs exist, so that might be worth investigating. When the problems first arose, the map drawing code was already quite dense so any change would require a huge overhaul. This meant that the original API was kept during the project, but if I return to it later I could test out some different APIs and compare performance.

Lag was also very noticeable when generating larger normalisation graphs. For these graphs I decided that it was worth adding the option to remove edge arrows – for linear terms, paths always travel downwards so arrows are not necessary. This can make the graph confusing for pure terms with paths that travel horizontally – but the performance improvement when removing arrows often makes a huge difference.

9.2.2 Delays in generating large galleries and graphs

A much more notable issue is the delay when generating galleries. This can range from a few seconds to not completing before the browser runs out of memory. While this was to be expected for large galleries, it is unfortunate that the delays become so large so soon (even generating the closed linear terms of size 11 can take over 30 seconds). There are two main reasons for this. As mentioned before, drawing the maps is quite performance intensive, and drawing a map for each portrait in the gallery is even more so – this is why the option to disable map drawing was added.

Another reason for the delay is the sheer number of terms that exist for larger galleries – it increases dramatically for small increases of n. For example, there are 1105 closed linear terms of size 11 but 27120 closed linear terms of size 14. This means that the generation algorithm has to find all these different terms, which can take an extremely long time. Even the original HASKELL implementations of the λ-term counting functions start to take a long time for values of n over 11.

There could be a solution to this if a more efficient algorithm is developed. The current algorithm recursively computes all the smaller subterms that could make up the abstractions and applications of the size specified before concatenating these together. It is possible that the algorithm could be made tail recursive to improve efficiency. For linear terms, there is also the problem of splitting the free variables between the LHS and RHS of an application, which was implemented quite hastily early on in the project. Perhaps by saving the results of these computations rather than recomputing them at each recursive call, the algorithm could be made more efficient.

There is a similar problem with very large normalisation graphs (with hundreds of nodes and thousands of edges, such as the graphs representing the boolean circuits). In these graphs, the number of paths is so large that even computing every single reduction node is impossible without running out of memory. Again, this could be due to an inefficiencies in the initialisation algorithm (such as how duplicate reduction nodes are found). Drawing the maps for each node also had a large performance impact, so the option to disable these was added. For graphs that could be generated in a suitable time, it was found that generating the path stats could still cause the browser to crash if there were enough different paths. This was changed to be an optional feature so that the graphs could still be viewed.

Unfortunately, these delays significantly reduce the potential scope of the tools. However, there is still plenty that can be done with smaller terms. I have only tested these delays on my desktop and laptop computer, so by using a more powerful system (perhaps renting a supercomputer) it is possible that these larger terms could be processed in a satisfactory time.

9.3 Comparison with other visualisation software

As mentioned in the introduction, there do not appear to be other pieces of software dedicated to representing λ-terms as rooted maps – so for this purpose these tools would be the clear choice. However, there is of course plenty of overlap with existing visualisation tools.

There seem to be two main paradigms in λ-term visualisers. The first are more traditional visualisers that stay close to the abstract tree structure, such as Ruiz and Villaret (2009) or Thyer (2007). The other paradigm includes the visualisers that take a new direction, such as the nested style used by Massalõgin (2008) and Citrin et al. (1995). Since the rooted maps are effectively abstract syntax trees with more restrictions on the structure, my visualiser is clearly the former, although it would be interesting to see how patterns in different maps generated by my visualiser translate to the less conventional approaches!

β-reduction is one of the main features in many visualisers. These are sometimes more polished than my implementation, as I spent more time on perfecting the visualisation before moving onto the reductions. One feature included in some software that my visualiser does not is smooth animation of reductions. This makes following the reduction easier – for the representation as a map it would be nice to see the appropriate application node replaced by the two edges representing the substitution of the argument into the function. The bubble representation used by Massalõgin (2008) animates this quite well, with function bubbles absorbing argument bubbles before 'popping' to represent the removal of the abstraction. Another example is the GoI visualiser by Cheung (2018), which can generate graphs representing the λ-term underlying functional programs. It uses a 'token' to traverse this graph and rewrite it appropriately to evaluate functions. The reduction can be viewed step-by-step, much like my visualiser, but the steps are more explicit – the substitution step is shown in addition to the finished reduction as in my visualiser. This could be a future direction for my visualiser to take.

Generation of normalisation graphs is less common, although implementations do exist. Grathwohl et al. (2011) can generate some very attractive graphs using several different algorithms, although it seems to be more concerned with the structure of the graph than the terms at each node. This implementation also omits self-loops 'for aesthetic quality', so the graphs generated will not necessarily be complete, unlike the graphs generated by my visualiser. It also does not compute statistics about the path lengths, which can be useful when considering complexity.

Let bindings (which I referred to as aliases in my project) were also used in many implementations. It was interesting to see the amount of thought that went into the use of 'macros', as they were called in Bharadwaj (2017), compared to my simple approach to aliases being additional labels on terms. The difference is that the former only expands macros when needed to at run time, whereas my visualiser expands them during parsing. Since my visualiser is concerned with the structure of the entire term, I believe my approach is more appropriate for displaying the maps.

There do not seem to be any tools available for displaying galleries of λ-terms, although there has been work on counting and generating them (Grygiel and Lescanne, 2013). Therefore this tool should be a useful addition to a researcher's arsenal. One interesting point is that the aforementioned paper by Grygiel and Lescanne (2013) provides many additional generation functions that could be added to the tools to expand the number of galleries available.

A common theme I experienced when exploring these tools was how many needed dependencies and used different languages. Of course, it may be likely that experienced researchers will already have these installed, but my visualiser does have the benefit of running in any modern browser with little fuss. This means even people with less experience can still play with the visualiser, which may be a springboard to learning more about the field.

9.4 Future development

9.4.1 Polishing the layout

With more development time, the way the tools are displayed could receive some more polish. The current layout is simple and functions well, but there are some annoying flaws. For example, some term names can spill onto multiple lines which can affect the rest of the page's layout. The pages do not display perfectly on every monitor either, since design was mostly catered towards the two machines I developed the code on. This would be a case of carefully adjusting the CSS of the pages, and adjusting how the font size is selected so it changes size dynamically based on how long the term is.

9.4.2 Experimenting with APIs

As mentioned above, it would be worth experimenting with different graph drawing APIs to see if any of them have better performance with larger maps. Another API that is widely used is GraphViz (`https://www.graphviz.org/`). Since this would require a large rehaul of the codebase, implementations using a different API could be carried out on a different Git branch.

9.4.3 Smoothly animated reductions

While the normalisation of terms can be animated, the reductions themselves are not – the original map is simply replaced with the new map. There is animation functionality in Cytoscape, although I have not investigated it much. Animating redexes would make it easier to follow how the map changes (e.g. showing how nodes are replaced with edges in a β-reduction).

9.4.4 Better display of normalisation graphs

As noted in the Implementation section, some normalisation graphs can be confusing when redexes travel horizontally to a node. This could be fixed by ensuring that nodes are placed in such a way that redexes are always travelling down the page. This would also make arrows less necessary and they could be omitted to increase performance.

9.4.5 Rehaul map-drawing algorithm

While the map-drawing algorithm is very successful and generates pretty maps, the code behind the scenes can get confusing at times. For example, the current algorithm still requires variable names to be specified for abstractions, which caused problems when integrating the visualiser with the gallery (as mentioned in Section 6). With more time, the algorithm could be refined so that it is simpler to follow.

9.4.6 Improve generation functions

There may be more efficient functions to generate λ-terms – for example, it may be possible to make the current functions tail recursive. This might make it possible to generate larger galleries.

There may also be more efficient ways to generate terms that satisfy some criteria. Currently, all terms for a given n and k are first generated, and then the ones that do not satisfy the specified criteria are removed from the list. This is incredibly inefficient when only a few terms need to be shown out of a large gallery (e.g. the 14/1105 terms of size 11 that contain 3 β-redexes), since lots of time is spent generating the complete set only to be immediately discarded.

Finally, there could be many other subsets of the λ-calculus to generate, such as all bridgeless terms (terms with no closed subterms).

9.4.7 Combinatorial maps

Another potential area to investigate identified in the Proposal was representing the λ-terms as *combinatorial maps*, as explained in Zeilberger (2016). The combinatorial representation of the generated map could be displayed alongside the visualisation.

10 Project management

10.1 Project structure

At the start of the project, I did not have much of a structure to the project other than 'implementing the λ-calculus'. Writing the Project Proposal helped me identify the core parts to the project: the visualiser, the gallery, and normalisation functionality (e.g. normalisation graphs). The project then took on a linear structure, working towards each of these overarching goals in turn. Although the Proposal gave some guidance as to the main features of each section, how these would be implemented was not specified so this was generally made up as I went along. This was probably not the best way to work in hindsight, as this meant that problems that cropped up due to not planning ahead with implementations were harder to solve since I had to unpick code written several weeks ago. Fortunately, I managed to overcome these problems and the end project is not too dissimilar to what is described in the Proposal – no major changes had to be made.

Once each part of the project had been completed, it would be rigorously tested to make sure that it worked satisfactorily before moving on to the next section. This was so that future areas of the project could use functions from previous modules without hassle (e.g. using the visualised maps in the gallery). This did not always go as well as hoped, as oversights caused code to be less flexible than intended. For example, when implementing the visualiser it originally expected predefined variable names (this would always be the case from user input), whereas the generator did not supply these, causing glitches in the gallery maps.

Initially the code for the visualiser and gallery interfaces was kept separate, other than the gallery accessing the map-drawing functions. However I realised when working on the portraits in the gallery that it would be better to share functionality between the two tools – this had the added benefit of reducing the size of the codebase. The result of this was that when the normalisation functionality was created later, it could be 'slotted in' to both tools while only having to be coded once.

While good progress was made most weeks and the end result was successful, I still feel that it would have been better to have a more thorough plan with explicit implementation goals for each week. Instead (especially in the early stages), goals for each week were quite loose and my work didn't have much structure other than 'This bit will be coded at some point'. This may have been because I was still unsure of where the project was heading, or how all the different components might work together. Fortunately as time progressed I started to get a better idea of the direction the project was taking – during the second term I was more confident of what I was going to do each week and as such progress was faster. Making a detailed plan would also have meant that future integration may have been smoother, since I would know exactly which functions would be used by future components and how.

10.2 Project log

A weekly project log was made on Canvas – these can be found in Appendix B. These turned out to be a good way of summing up what had been achieved in the week and gave an indication of what might need to be worked on next. The logs also provided a good record of how the project evolved over time.

10.3 Supervisor meetings

Meetings with the Project Supervisor were held once a week (when possible), to demonstrate what had been achieved in the previous week, and discuss any issues that had arisen. These discussions often included where the project would be heading next, and would cover the background knowledge required for some of the features (such as developing functions for counting and generating λ-terms). The meetings were also useful for discussing the aims of the project and how it might be used.

10.4 Version control

Throughout the project, Git was used to maintain version control. This was useful when implementing new features – if a new bug appeared an old commit could be checked out to find what might have introduced it. Git was also a useful companion to the progress log when reflecting back on my progress throughout the project, as my commits often provided useful information about the exact code that was worked on in which week. In future though, I might try and make my commit names more descriptive since `Fixed the bug` or similar isn't very useful several weeks afterwards!

11 Conclusion

Overall, the λ-term visualiser and λ-term gallery should be very useful tools for researchers interested in examining the connections between λ-terms and rooted maps in the future. With the visualiser, users can quickly generate the maps for any λ-term, and the gallery can be used to generate terms that satisfy various criteria. While performance issues do exist for larger terms, I feel the project has still been successful in its aims as the tools are still capable of generating many terms and galleries. As no tools focusing on the representation of λ-terms as rooted maps currently exist, I believe that these tools offer a new way to aid research in this area, reducing the dull elements and making the process far more efficient.

References

Bharadwaj, C.
2017. Lambdalab: Interactive λ-calculus for learning.

Bodini, O., D. Gardy, and A. Jacquot
2013. Asymptotics and random sampling for bci and bck lambda terms. *Theoretical Computer Science*, 502:227–238.

Cheung, S.
2018. Goi visualiser. https://cwtsteven.github.io/GoI-Visualiser/CBV-with-CBV-embedding/index.html. Accessed on 07/04/2019.

Church, A.
1936. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363.

Church, A. and J. B. Rosser
1936. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482.

Citrin, W., R. Hall, and B. G. Zorn
1995. Programming with visual expressions. In *VL*.

Erwig, M.
1998. Abstract syntax and semantics of visual languages. *Journal of Visual Languages & Computing*, 9(5):461–483.

Grathwohl, N. B. B., J. Ketema, J. D. Pallesen, and J. G. Simonsen
2011. Anagopos: A reduction graph visualizer for term rewriting and lambda calculus. In *22nd International Conference on Rewriting Techniques and Applications (RTA'11)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Grygiel, K. and P. Lescanne
2013. Counting and generating lambda terms. *Journal of Functional Programming*, 23(5):594–628.

Kaye, G. J.
2018a. Final year project - proposal.

Kaye, G. J.
2018b. A visualiser for linear λ-terms as rooted 3-valent maps.

Lando, S. K. and A. K. Zvonkin
2013. *Graphs on surfaces and their applications*, volume 141. Springer Science & Business Media.

Mairson, H. G.
2004. Functional pearl linear lambda calculus and ptime-completeness. *Journal of Functional Programming*, 14(6):623–633.

Massalõgin, V.
2008. Visual lambda calculus. Master's thesis, University of Tartu.

Pierce, B. C.
2002. *Types and programming languages.* MIT press.

Ruiz, D. and M. Villaret
2009. Tilc: The interactive lambda-calculus tracer. *Electronic Notes in Theoretical Computer Science*, 248:173–183.

Statman, R.
1974. *Structural complexity of proofs.* PhD thesis, Stanford University.

Thyer, M.
2007. Lambda animator. Accessed on 07/04/2019.

Victor, B.
2007. Alligator eggs! a puzzle game. Accessed on 07/04/2019.

Wadsworth, C. P.
1971. *Types and Programming Languages.* PhD thesis, University of Oxford.

Zeilberger, N.
2016. Linear lambda terms as invariants of rooted trivalent maps. *Journal of Functional Programming*, 26.

Zeilberger, N.
2018. Lambda calculus and the four colour theorem. The Oxford Advanced Seminar on Informatic Structures, 22 June 2018.

A File structure

This appendix details the file structure of the submitted ZIP directory. To run either of the tools, simply load up one of the `.html` files in the root of the directory in your preferred browser.

A.1 `root` – Main files

- `visualiser.html`: The λ-term visualiser.
- `gallery.html`: The λ-term gallery.
- `aliases.txt`: Some sample aliases for use in the visualiser, including the boolean circuit functions, adapted from Mairson (2004).
- `README.md`: A readme file.

A.2 `docs` directory – Documentation

- `2018-10-26-project-proposal.pdf`: The original project proposal.
- `2018-11-23-scientific-paper.pdf`: The scientific paper, detailing some project background and the state of the project towards the end of November 2018.
- `2019-04-08-final-report.pdf`: The final report (this document).

A.3 `src` directory – Source code

- `cytoscape.min.js`: The source code for the graph drawing API used.
- `definition.js`: The implementation of the λ-calculus and associated data structures.
- `evaluator.js`: Functions to evaluate and normalising λ-terms.
- `galleryFunctions.js`: Functions used in the gallery.
- `Generator.hs`: The original Haskell versions of the λ-term counting and generating functions.
- `generator.js`: Functions to count and generate λ-terms.
- `graph.js`: Functions to generate the λ-term maps and normalisation graphs
- `parser.js`: Functions to parse and tokenise user input into λ-terms.
- `sharedFunctions.js`: Functions shared between the gallery and visualiser.

- `style.css`: The style document for the two HTML pages.
- `visualiserFunctions.js`: Functions used in the visualiser.

A.4 `pics` directory – Images

Some images of the tools in action.

A.5 `tests` directory - Unit tests

- `parser`: Inputs used for unit testing the parser, with the corresponding de Bruijn notation for valid inputs.
- `visualiser`: Images comparing hand-drawn maps for terms with the corresponding maps generated by the visualiser.
- `gallery`: Sample outputs for some small galleries, to be compared against the outputs generated by the program.
- `normalisation`: Images comparing hand-drawn normalisation graphs with the corresponding graphs generated by the visualiser.

B Project log

Week 1 Explored different directions the project could take. Started implementation of λ-calculus in JAVASCRIPT.

Week 2 Completed λ-term parser. Implemented shifting and substitution on λ-terms. Investigated graph theory JAVASCRIPT libraries.

Week 3 Implemented the basics of graph rendering in the visualiser. Identified project scope as investigating different subsets of the λ-calculus.

Week 4 Wrote up the project proposal.

Week 5 Ensured that the order of edges around nodes in the visualiser was correct. Corrected labels in the visualised map.

Week 6 Completed normalisation function for λ-terms. Looked into enumeration and generation algorithms for pure, linear and planar λ-terms. Started writing the scientific paper.

Week 7 Looked into making the visualised λ-terms display correctly and clearly. Finished first draft of scientific paper.

Week 8 Finished scientific paper. Continued tinkering with how the visualiser displayed maps.

Week 9 Completely refactored the map drawing code. Changed to a new strategy of drawing maps with variables at the top.

Week 10 Added free variables to the visualised maps. Implemented enumeration and generation functions in HASKELL.

Week 11 Created basic gallery page.

Christmas Reworked map drawing to look up variables in a context rather than using variable names. Completed the basic gallery page for generating pure, linear and planar terms.

Week 12 Investigated how to calculate crossings in λ-terms.

Week 13 Finished implementing crossings algorithm. Created bigger λ-term 'portraits'.

Week 14 Adjusted crossings algorithm to make it tidier. Started implementing normalisation functionality in the gallery.

Week 15 Implemented reducing the visualised map to its normal form one step at a time.

Week 16 Implemented displaying all β-redexes in a term. Implemented highlighting redexes in λ-terms.

Week 17 Added regular notation in addition to de Bruijn in the gallery for easier reading. Implemented normalisation graphs.

Week 18 Switched to using images rather than compound nodes for normalisation graphs. Implemented highlighting redexes in the visualised map. Brought features from the λ-term portraits into the main visualiser.

Week 19 Switched normalisation graph implementation from a tree to an adjacency matrix. Added feature to define aliases in the visualiser. Adjusted how variable naming is used.

Week 20 Added 'bulk aliases' function to paste in many aliases at once. Made sure loop edges rendered in normalisation graphs. Added ways to improve performance such as disabling edge arrows. Added full screen mode to visualiser. Added animated normalisation feature.

Week 21 Prepared for demonstration. Added feature to hover over edges in normalisation graphs to view redex.

Week 22 onwards Wrote up final report.