Under consideration for publication in J. Functional Programming

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

George Kaye
1522391
Supervisor: Noam Zeilberger

Abstract

This paper details a project to create a visualiser to display λ-terms as maps. This visualiser will be a useful tool in enabling researchers to better relate the computational and logical properties of λ-terms with the topological properties of maps in graph theory. In particular, this paper focuses on the way this visualiser can be used to make some headway on the open problem of normalisation of planar λ-terms. It gives a brief background on the associated topics of the λ-calculus and rooted maps, as well as describing some of the research and problems associated with this topic. It also shows the current process of the visualiser and details the next steps that will be taken in the project.

1 Introduction

The pure λ-calculus has links to many different areas of mathematics and computer science. One such area is graph theory – the linking of the computational and logical properties of λ-terms with topological properties of rooted maps.

To explore these links, I have begun a project to create a visualiser to display various λ-terms (and in particular, linear terms) as rooted maps. This visualiser can then be used as a helpful tool for performing experimental mathematics with λ-terms, such as exploring different fragments of the λ-calculus. To help with this, the tool will also contain a generator of various λ-terms. The visualiser could then be used (by myself or others) to study lots of interesting problems.

One possible area to explore (and one studied in greater detail in this paper) is the open problem of normalisation of planar λ-terms (which relate to planar maps). While it is known that computing the normal form of pure λ-term is undecidable (Church, 1936), and that the upper bound of complexity of normalising linear λ-terms is in polynomial time (Mairson, 2004), it is unknown if there is a lower bound on the complexity of normalising planar λ-terms. The visualiser could be used to generate maps of terms in normalisation graphs, hopefully making some headway on this problem.

The paper begins by providing details on the relevant concepts of λ-calculus (Section 2) and graph theory (Section 3), before combining these to discuss the representation of linear terms as *rooted 3-valent maps* (Section 4). Section 5 details the problem of normalisation in more detail, and Section 6 talks about enumerating and generating λ-terms. Details about the current project status and its future scope are found in Sections 7 and 8.

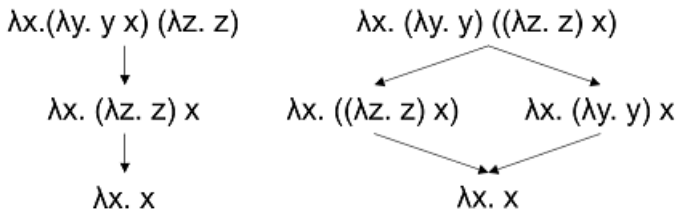

Fig. 1. An example of two normalisation graphs, showing the steps of reduction taken for the terms $\lambda x.(\lambda y\,x)(\lambda z.z)$ and $\lambda x.(\lambda y.y)((\lambda z.z)\,x)$ to reach their normal forms. Note that the diverging paths in the right example still lead to the same normal form, as stated by the Church-Rosser theorem.

2 A refresher on the pure λ-calculus

The λ-calculus is a model of computation where programs are represented by variable abstraction and function application. Developed by Alonzo Church in the 1930s (Church, 1932), it has become the basis for all functional programming languages. Any term in the λ-calculus (a λ-term t) can be defined as one of the following:

Abstraction	$\lambda x.t$
Application	$t_1\,t_2$
Variable	x

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 (it is a local variable), or free otherwise. 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*.

Computation in the λ-calculus is performed by applying *reduction operations*. The two main operations are:

α-conversion	$\lambda x.T \to_\alpha \lambda y.T[x \mapsto y]$	Renaming variables in a context
β-reduction	$(\lambda x.T)\,u \to_\beta T[x \mapsto u]$	Applying a function to its argument

The idea of repeatedly applying reductions to a λ-term is known as *normalisation*. A term that can be β-reduced is known as a *β-redex* – in normalisation β-reduction is repeatedly applied until a *normal form* containing no β-redexes is reached. However, as we will see later, this is not always computable. When there are multiple redexes in a λ-term, the order they are reduced in does not matter – reducing either redex will lead to the same normal form, providing the normalisation terminates. This is known as the *Church-Rosser theorem* (Church & Rosser, 1936). We can represent normalisation of terms with *normalisation graphs*, that show the reductions taken to reach a normal form. An example of two normalisation graphs is shown in Fig. 1.

α-conversion is essential when normalising terms with explicit labels to avoid a collision after β-reduction. For example: $\lambda x.(\lambda y.\lambda x.y\,x)\,x \to_\beta \lambda x.(\lambda x.x\,x)$, where the two x variables refer to different abstractions, but we don't know which one! This can be resolved by performing α-conversion on the initial term: $\lambda x.(\lambda y.\lambda x.y\,x)\,x \to_\alpha \lambda x.(\lambda y.\lambda z.y\,z)\,x \to_\beta \lambda x.(\lambda z.x\,z)$.

This process can be inefficient to implement. To avoid this problem, we can represent terms using *de Bruijn indices*. Rather than explicit labels, variables are numbered according

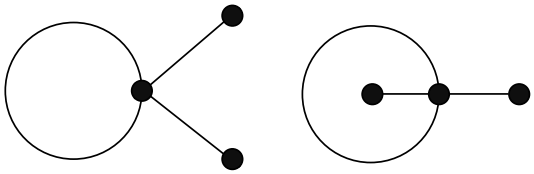

Fig. 2. These two diagrams represent the same graph but two distinct maps (the ordering of edges around the point on the circle is changed). Adapted from Lando, Zvonkin (2013)

to how many abstractions one must pass through to find the term it represents. For example, $\lambda x.\lambda y.\lambda z.xyz$ becomes $\lambda.\lambda.\lambda.210$. This removes the need for α-conversion altogether!

When looking at λ-terms, we can talk about the number of *subterms* a term has. This is defined as:

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

The pure λ-calculus contains all the terms made by combining abstractions, applications and variables. However, by placing restrictions on how terms are formed, one can explore many different *fragments* of the pure λ-calculus. The *linear* λ-calculus contains all terms in which variables are used exactly once – an example is the term $\lambda x.\lambda y.\lambda z.x\,(zy)$. The linear λ-calculus can be useful for considering problems such as resource limitation. Linear terms can also be *planar* if variables are used in the order they were abstracted, such as the term $\lambda x.\lambda y.\lambda z.x\,(yz)$. Linear and planar λ-terms have interesting properties related to *rooted maps*.

A more in-depth definition of the pure λ-calculus can be found in Barendregt (1984).

3 3-valent rooted maps

A graph is a set of nodes and edges (which can be undirected or directed) connecting pairs of nodes. These graphs are quite 'abstract': the actual positioning of the nodes and edges is not important, so long as the *adjacency relation* (which nodes are connected to each other) is preserved. The idea of representing λ-terms as graphs is not new - Wadsworth (1971) and Statman (1974) studied it for their PhD theses. However, more recent research has uncovered links between linear λ-calculus and the combinatorics of *rooted 3-valent maps* (Bodini *et al.*, 2013). We shall explore these maps in this section.

When these graphs are drawn onto surfaces (or *embedded*), they are given a physical structure which is called a *map*. Embedding a graph does not always result in the same map, as can be seen in Fig. 2. The adjacency relations are the same for both maps, but the *faces* bordered by the edges are different (in an abstract graph, there are no faces). This is due to the *cyclic order* of edges around a node - an important distinguishing point between graphs and maps. We can represent the orders of edges around nodes and the faces of the map as *permutations* in a *combinatorial map* - this is discussed in Zeilberger (2016).

A map also has a *genus*. This is the lowest amount of 'holes' that a surface can have in order for the graph to be embedded in a surface. For example, a *planar* map (with no

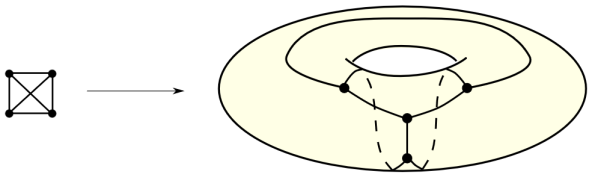

Fig. 3. An example of how a graph with crossings can be embedded onto a torus. From Zeilberger (2018).

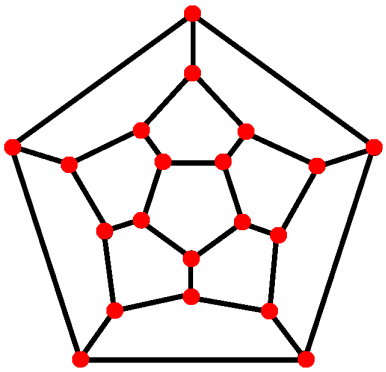

Fig. 4. Another example of a planar map. From https://en.wikipedia.org/wiki/Planar_graph

crossing of edges) can be embedded into a sphere - a surface with genus 0. A map with one crossing can be embedded into a torus - a surface with genus 1 (an example of this can be seen in Fig. 3). The number of crossings is not necessarily equal to the genus - the only link is that the number of crossings is greater than or equal to the genus. For all maps, we can use the genus to compute *Euler's characteristic*, $\chi = 2-2g$ (Lando & Zvonkin, 2013). We can check this for the map in Fig. 4. It has 10 vertices, 20 edges, 12 faces and a genus of 0:

$$v-e+f = 2-2g \Longrightarrow 10-20+12 = 2-0$$

When considering the links between λ-terms and maps, we must consider *rooted 3-valent maps*. A *3-valent* map is a map where every vertex has three edges that connect to it (i.e. the vertices all have a *valency* of 3). If we then add a 'special' vertex at some point (the

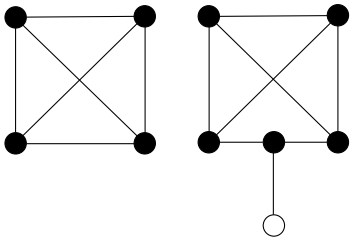

Fig. 5. A 3-valent map, and a the same map but rooted (root indicated by the white node)

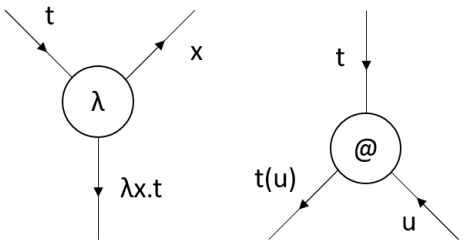

Fig. 6. An abstraction and an application, represented as nodes of a map.

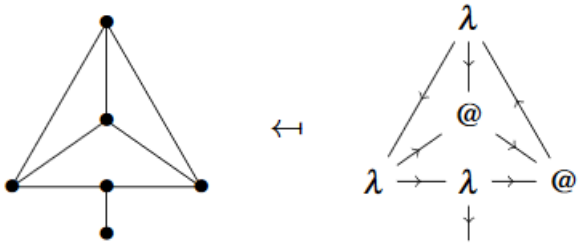

Fig. 7. A representation of the term $\lambda x.\lambda y.\lambda z.x\,(y\,z)$ as a rooted 3-valent map, without and with node labels. From (Zeilberger, 2016).

root of the map) that connects to the map at one point, we form a *rooted map*. An example of how this works is shown in Fig. 5.

4 Linear λ-terms as rooted 3-valent maps

To represent λ-terms as rooted maps, we represent abstractions and applications as nodes, as shown in Fig. 6. We can think of the ordering of the edges around nodes 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 denote the start of the term, these nodes can be combined to create a rooted map, as shown in Fig.7. For a linear term, this map will always be 3-valent since there is exactly one use of abstracted variables.

Viewing these maps of λ-terms can be useful for many reasons. We can easily tell which terms are planar, since these terms produce planar maps (with no crossings). These maps

also allow us to inspect the combinatorial properties of the terms, such as by using Euler's characteristic.

5 Normalisation of terms

One area of problems that could benefit greatly from this visualisation of terms is the normalisation of λ-terms. While all λ-terms have a normal form, the problem of computing the normal form is less trivial – some terms never terminate. This is because upon normalising, terms with repeated variables may not reduce in size. One famous example is the term Ω:

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

Attempting to β-reduce the term Ω simply results in a return to the original expression! Therefore any attempts to normalise Ω will loop forever. This is not immediately obvious from looking at the term, so is it possible to decide if computing a normal form will terminate?

5.1 Deciding if normalising a pure λ-term will terminate

Church (1936) showed that this problem is in fact undecidable. He did this by showing that the λ-calculus can be used to simulate a Turing machine. For example, integers can be represented as *Church numerals*:

$$0 = \lambda f.\lambda x.x$$
$$1 = \lambda f.\lambda x.fx$$
$$\mathbf{succ}\, n = \lambda f.\lambda x.f^{n+1}x$$

Recursion can also be defined using a *fixed-point combinator* (an expression f such that $f(x) = x$). One such example is Turing's fixed-point combinator:

$$\Theta = (\lambda x.\lambda y.y(xxy))(\lambda x.\lambda y.y(xxy))$$

Since we can encode the λ-calculus as a Turing machine, the problem of computing a normal form (and completing execution) is reduced to the halting problem (which has been proven to be undecidable) - so the problem itself cannot be decidable.

5.2 Complexity of normalising linear λ-terms

However this proof does not hold for linear λ-terms, because Turing's fixed-point combinator is not a linear term. Fortunately, it turns out that that computing the normal form of a linear λ-term is decidable, and its complexity has an upper bound of polynomial time. This section will briefly explain why this is the case.

To check if normalising a linear λ term with n subterms will terminate, we first have to perform a linear search 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. This means that terms cannot 'blow

```
fun True x y = Pair x y;
val True: fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c

fun False x y = Pair y x;
val False: fn : 'a -> 'b -> ('b -> 'a -> 'c) -> 'c

fun Not P x y= P y x;
val Not = fn : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
```

Fig. 8. The boolean logic constructs `True`, `False` and `Not` encoded in `ML` for linear λ-terms, given by Mairson (2004). `True` and `False` are encoded as pairs, where the first element is the 'real' value and the second value is a garbage value. This ensures that each argument is used exactly once in each function, satisfying the linearity condition.

up' and loop infinitely, like Ω. Knowing this, we can use the definitions of $subterms(t)$ from Section 2 to write

$$subterms((\lambda x.t)\,u) = 1 + (1 + subterms(t)) + subterms(u)) \quad (1)$$

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

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 $\frac{n}{3}$. Of course, there are many terms that take fewer steps to find a normal form, such as terms already in normal form (e.g. $\lambda x.x$).

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 is in $\mathcal{O}(n^2)$ – polynomial time.

5.3 PTIME-completeness for normalising linear λ-terms

Not only is normalising a linear λ-term computable in polynomial time, the problem is also PTIME-complete (Mairson, 2004). This is interesting as PTIME contains problems that are often not easily parallelisable (Greenlaw *et al.*, 1995), which corresponds with how λ-terms may have multiple (parallel) paths through a normalisation graph to reach their normal form.

To prove PTIME-completeness for linear λ-terms, Mairson (2004) wrote several constructs in `ML` to enable the encoding boolean circuits as λ-terms. Some of his functions are shown in Fig. 8. Since the Circuit Value Problem (in which we compute the result of a boolean circuit given its inputs) is known to be a PTIME-complete problem, by reducing it to normalising linear λ-terms (the underlying basis for `ML`), this proves that the normalisation of linear terms is also PTIME-complete.

5.4 Normalisation of planar terms

Once again, we run into a problem if we attempt to use this proof for the normalisation of planar λ-terms. We can see why in the examples in Fig. 8 - the arguments in Mairson's functions are not necessarily used in the order they are introduced!

Because all terms of the planar λ-calculus are themselves linear (each argument is still used once), the upper bound of normalising these terms is the same as for linear terms – polynomial. However, there may be a lower bound of complexity for normalising planar λ-terms that is unknown to us.

This is just one of the problems that the visualiser might be able to help us with. For example, we could generate maps of the terms at each level of a normalisation graph, and see what interesting properties we can derive from them. We would not necessarily have to limit ourselves to planar terms either – one alternative is to study fragments of the λ-calculus that produce maps of different genus and see how normalisation is affected.

6 Enumerating and generating λ-terms

One thing we can do to help us investigate these various fragments of the λ-calculus is to *enumerate* and *generate* terms from them. Surprisingly, not much work on this area has been carried out (Grygiel, 2013). However, there has been plenty of work counting first order terms without bound variables, with work going as back as far as Hipparchus of Rhodes (Flajolet & Sedgewick, 2009).

6.1 An example

A good starting point is to generate λ-terms with a given number of *subterms* and *free variables*. We can enumerate the number of general λ-terms with subterms $n > 0$ and free variables k inductively:

$$enum(n,k) = enum(n-1,k+1) + \sum_{n_1=1}^{n-2} enum(n_1,k) \cdot enum(n-1-n_1,k) + [n=1]\,k$$

The three terms correspond to abstractions, applications and variables respectively. The number of abstractions can be calculated by enumerating 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!

It is fairly straightforward to develop this equation into a program for enumerating λ-terms. An example in `Haskell` is shown in Fig. 9. An enumeration program can also be easily modified to *generate* a list λ-terms, as shown in Fig. 10

The number of enumerated terms of different subsets of the λ-calculus often form known sequences. The number of closed λ-terms (generated by `[enum n 0 | n <- [0..]]`) forms the sequence `[0,0,1,2,4,13,42,139,506,1915,7558]`: sequence `A135501` on the Online Encyclopedia of Integer Sequences (OEIS). Different fragments such as linear and planar λ-terms form different sequences (Zeilberger, 2016).

```
enum :: Int -> Int -> Int
enum 0 _ = 0
enum 1 k = k
enum n k = enum (n-1) (k+1)
           + sum [(enum n1 k) * (enum (n-1-n1) k) |
                                       n1 <- [1..n-2]]
```

Fig. 9. A program to enumerate pure λ-terms of a given number of subterms and free variables

```
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]
```

Fig. 10. A program to generate pure λ-terms of a given number of subterms and free variables

With some modifications related to how free variables are shared between subterms of applications, the programs above can be used to find just linear or planar terms. In the case of planar terms, the context of free variables is 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]` 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. The combinations of variables must instead be considered, using $\binom{n}{r}$ notation in the enumeration equation.

This is not the only way of enumerating λ-terms – Grygiel & Lescanne (2013) took a slightly different approach, using the *size* of a term rather than its subterms. This produces sequence `A220894` for numbers of closed terms of size n.

7 The project

This project brings all of these areas together. The primary goal is to develop a visualiser that can display λ-terms from user input as rooted maps. This will then be combined with a generator of λ-terms from different fragments, which will allow us to investigate topological properties of these fragments. Once these two components are completed, this visualiser could be used for many different problems, such as the problem of finding the lower bound of complexity on the normalisation of planar terms.

Some progress has already been made with developing the visualiser. This section will detail what has been done, and what is yet to come.

```
class LambdaVariable {
    constructor(x, label)
    ...
}

class LambdaAbstraction {
    constructor(t, label)
    ...
}
class LambdaApplication {
    constructor(t1, t2)
    ...
}

const var1 = new LambdaVariable(0, 'y');         // y
const var2 = new LambdaVariable(1, 'x');         // x
const app = new LambdaApplication(var2, var1);   // x y
const abs = new LambdaAbstraction(app, 'y');     // \y. x y
const abs2 = new LambdaAbstraction(abs, 'x');    // \x.\y. x y
```

Fig. 11. The constructors of the three types of λ-terms, implemented in `Javascript`, as well as an example of how the term $\lambda x.\lambda y.xy$ would be represented. The `label` variables are effectively optional – they are only used when printing the terms.

7.1 Implementing the λ-calculus

The first stage of the project was to implement the λ-calculus in a language of choice. A basic parser could then be developed to take user input and convert it to this internal representation of λ-terms, which could then be displayed on the screen.

The language chosen was `Javascript`, for the easy ability to run as a web app in a browser without having any dependencies. This will be useful if others were to use the tool after the project, and also means it will be easy to use on any machine.

The implementation was partially based on the `ML` examples developed by Pierce (2002). λ-terms are stored in de Bruijn notation – this removes the need to implement checks for α-equivalence – with an associated label for pretty printing. The constructors for the three implemented classes, along with an example, can be seen in Fig. 11.

The current visualiser interface can be seen in Fig. 12. A user can input a λ-term t and a context Γ, and view this represented as a map on screen. The map is generated using the Javascript library `Cytoscape.js`, which allows for fairly simple creation of maps by defining nodes and edge objects, and how they link together.

The term is also represented in de Bruijn notation for reference. Users can then evaluate or normalise the term. An outermost strategy is used for normalisation, which means that a normal form will be found if one is computable. To prevent infinite loops if this is not the case, evaluation or normalisation is restricted to a number of steps before the program gives up. This does mean that there are terminating terms that cannot be normalised by the program, but if the time-out parameter is set to sufficiently large value, it shouldn't be a critical problem. A user-configurable time-out could be implemented to allow more

execution steps if the machine can handle it (my laptop struggles even at 100 steps, but my desktop can handle it with ease).

7.2 Generating the map

Automatically generating correct maps has been the most difficult part of this stage of the project. While it is often simple to draw the maps by hand, making a generalised algorithm for all terms is significantly more challenging.

The current algorithm for drawing the maps is based off a tree structure combining the nodes displayed in Fig. 6. The algorithm takes a λ-variable and recursively converts into node and edge objects. For example, $(\lambda x.x)(\lambda y.y)$ would first generate an application node, then deal with the first term in the application, and then the second term. The general algorithm used to position the various nodes in Fig. 12 is as follows:

1. If this is the start of the term:
   - The root node is placed at the bottom of the screen.
   - The first node (corresponding with the outermost subterm) is placed directly above the root node.
2. The algorithm is then run recursively on the next subterm:
   - If the current subterm is an abstraction, the algorithm is run on the scope.
   - If the current subterm is an application, the algorithm is run on the two subterms sequentially.
3. The next node is placed depending on its 'parent':
   - If the parent was an application, it is placed up and to the left of the first node.
   - If the parent was an application and this is the first term, it is placed up and to the left of the first node.
   - If the parent was an application and this is the second term, it is placed up and to the right of the first node.
4. The algorithm terminates when it reaches a variable.

When a variable is encountered, an edge is drawn connecting where it is initially abstracted with where it is used. If the variable is free, an extra node is created to represent the external abstraction. In theory the connecting edge would be a clean curve, but this is simulated using a 'support node' to guide the edge towards the top of the screen so this new edge doesn't erroneously cross any other edges – examples can be seen as the small grey squares at the top of Fig. 12. A 'perfect' representation of the map with smooth curves is shown in Fig. 13, but the 'spiky' curves are still an acceptable way of drawing the map, since the faces and ordering of edges is preserved.

While this algorithm works well for some terms, it struggles with many others. One particular struggle is how the algorithm deals with complex subterms. This can be seen in Fig. 14, where the term $\lambda x.\lambda y.\lambda z.(\lambda a.ay)\,z$ is mangled particularly badly. This is due to the subterm of $\lambda a.ay$ being positioned in such a way that it intersects with the edges and nodes for the outer term. The problem is not unsolvable – Fig. 14 also shows how this term could be displayed correctly. The ultimate goal now is to develop an algorithm that

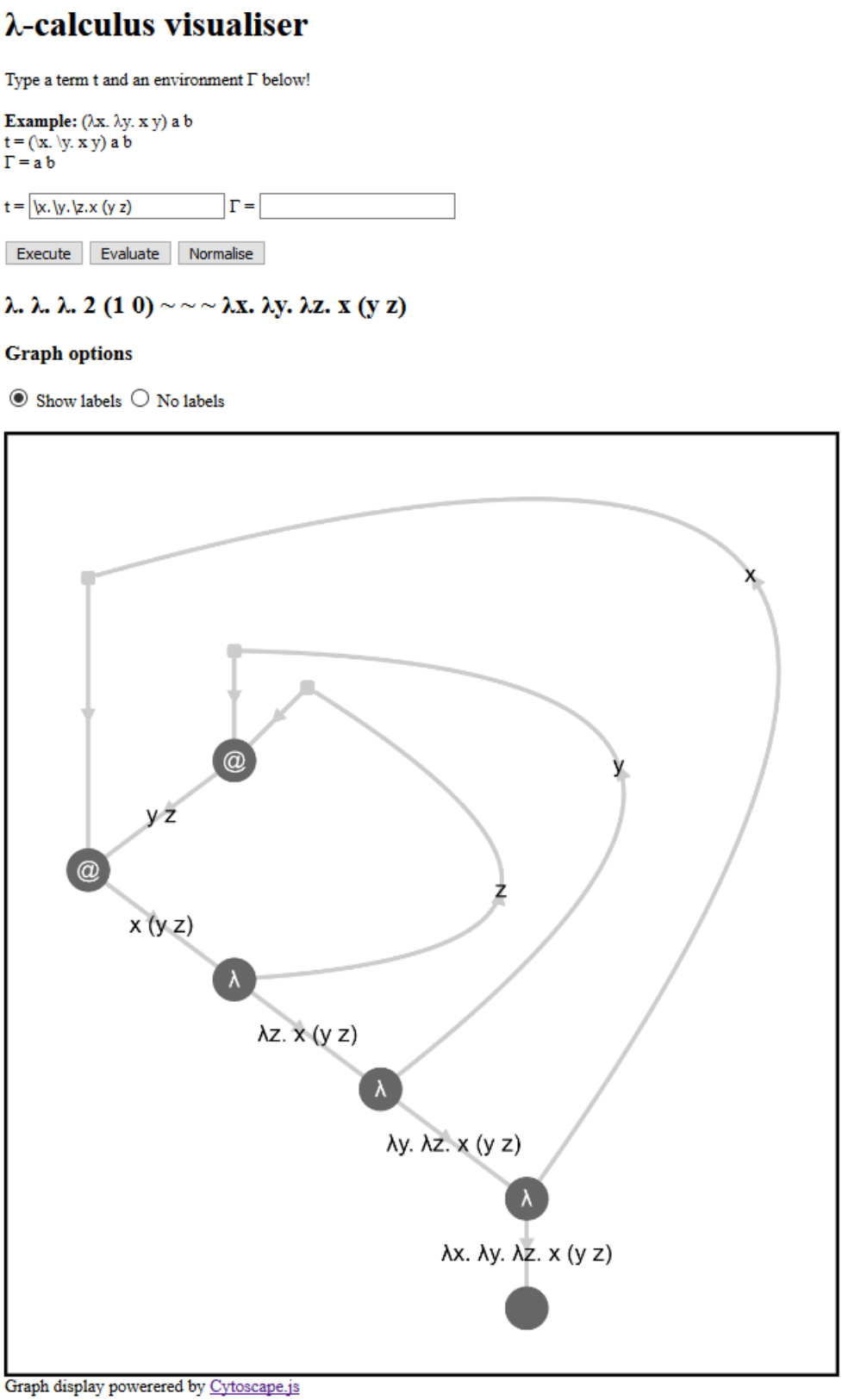

Fig. 12. An example of the visualiser generating the map for the term $\lambda x.\lambda y.\lambda z.x\,(y\,z)$

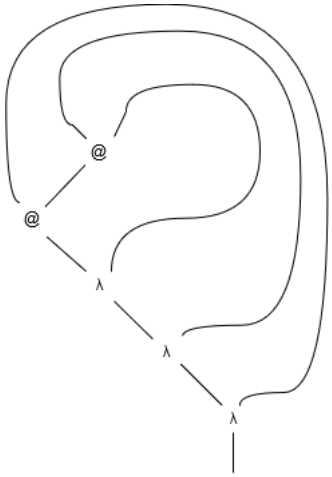

Fig. 13. A 'perfect' representation of the term $\lambda x.\lambda y.\lambda z.x\,(y\,z)$. Although it looks prettier than the representation in Fig. 12, both are the same map.

displays the term like this *automatically*, and does this correctly for all terms. This could be done by:

- Making sure support nodes are always placed above the node they come from for clarity (e.g. in Fig. 14, the support nodes in the top left corner skew the edges in confusing ways).
- Making sure subterms on the same 'level' (e.g. the two terms of an application) are kept far apart so their terms don't conflict.
- Computing the area taken up by subterms, and using this to place support nodes of conflicting edges far enough away from the term.
- Spreading the map out further, reducing the chance of edges colliding when they shouldn't (but this might make the map elements smaller and harder to read).

7.3 Criteria for a successful visualiser

To judge how useful the visualiser will be, there are several criteria that it must satisfy to be a successful tool:

- **Correctness:** The visualiser must generate maps that accurately represent the λ-terms the user has entered. Ordering of edges around nodes is especially important here, as different ordering of edges can lead to completely different maps. Sometimes the generated maps are not correct, as the way some edges are drawn breaks the cyclic ordering of the edges around nodes. This can be fixed by using the support nodes to guide curves in the right direction out of the relevant nodes.
- **Clarity:** The maps generated by the visualiser must be easy to read – the maps shouldn't be too cramped or too small to make out. `Cytoscape` supports zooming and moving around the generated maps, meaning that even large maps should be easy to investigate.

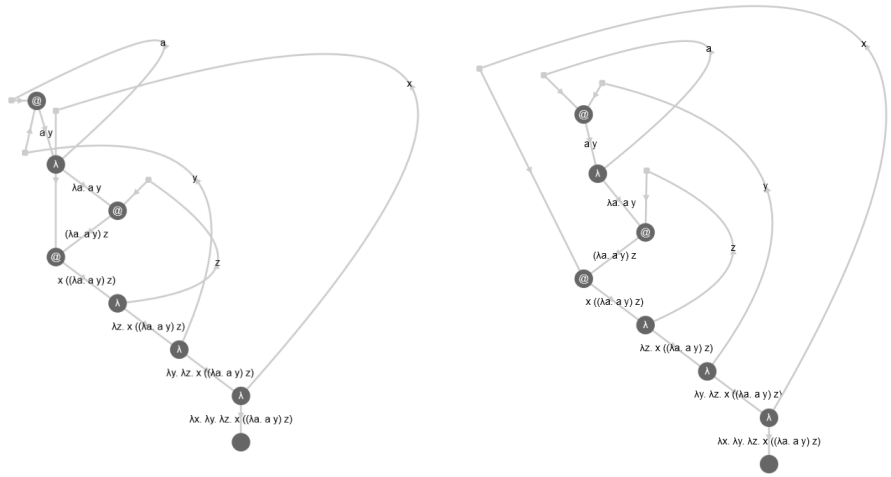

Fig. 14. The current generated representation of $\lambda x.\lambda y.\lambda z.x(\lambda a.ay)z$ (left), and a more successful representation (right), achieved by moving some of the nodes around by hand.

- **Completeness:** The visualiser should be able to generate maps for all λ-terms, not just a select few. Currently the visualiser can generate maps for all terms in some form.
- **Aesthetics:** A somewhat minor point compared to the other criteria, but it would be good if the generated maps were at least a bit pretty.

8 Future work

The main issue that needs to be sorted out before progress can be made is making sure the generated maps display correctly. One way this problem could be simplified would be to first try and generate the maps without the curved variable edges (i.e. as binary trees) – this would get the node structure down without all the edges getting in the way. Once we have successfully developed an algorithm for this, we could then try and integrate the variable edges.

Once we have the visualiser correctly generating maps, we can move onto other tasks. The next step will be to translate the algorithms from Section 6 into Javascript. This should not be too difficult as the programs are very simple, although the pattern matching of the `Haskell` programs will have to be translated into if statements, and list comprehensions replaced with for loops. Another task to do at this point would be to create similar algorithms for generating linear and planar terms, as described in Section 6.

We can then combine the visualiser and the term generator to create a 'gallery' of all terms of a certain λ-fragment on the screen. An example is shown in Fig. 15 for closed λ-terms with four subterms. This would enable us to investigate the topological properties of different terms within the same fragment. One option we could take is to generate combinatorial maps for all the generated terms, and compute interesting topological properties such as genus.

At this stage, we could use the visualiser to start looking at more specific problems. The problem we have focused on in this report has been the normalisation of terms. With the

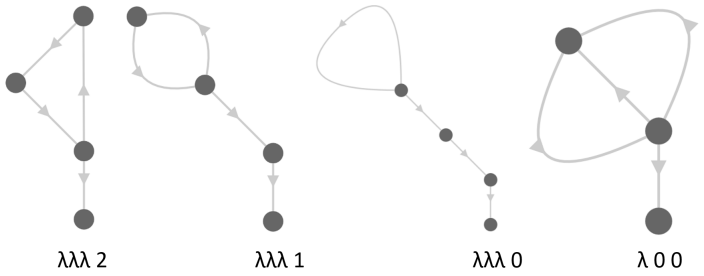

Fig. 15. All closed λ-terms with four subterms displayed as maps with the visualiser. Since these are not linear terms the maps are not 3-valent.

visualiser's normalisation function, we could generate normalisation graphs in addition to the maps of terms, creating a graph of maps. This lets us observe properties such as path length or whether there are multiple different routes to take. We could compare many different sets of these graphs and try to identify and patterns or peculiarities. For example, is there a link between term size and how long it takes to normalise?

There are many other areas that might be interesting to study other than normalisation. The typing of the generated maps could be studied, as this links to normalisation. There are also the topological properties of the generated maps to consider, such as the genus, diameter or connectedness – how do these translate to properties of λ-terms?

References

Barendregt, H. P., *et al.* . (1984). *The lambda calculus*. Vol. 3. North-Holland Amsterdam.

Bodini, O., Gardy, D., & Jacquot, A. (2013). Asymptotics and random sampling for bci and bck lambda terms. *Theoretical computer science*, **502**, 227–238.

Church, A. (1932). A set of postulates for the foundation of logic. *Annals of mathematics*, 346–366.

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

Church, A., & Rosser, J. B. (1936). Some properties of conversion. *Transactions of the american mathematical society*, **39**(3), 472–482.

Flajolet, P., & Sedgewick, R. (2009). *Analytic combinatorics*. cambridge University press.

Greenlaw, R., Hoover, H. J., Ruzzo, W. L., *et al.* . (1995). *Limits to parallel computation: P-completeness theory*. Oxford University Press on Demand.

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

Lando, S. K., & Zvonkin, A. K. (2013). *Graphs on surfaces and their applications*. Vol. 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.

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

Statman, R. (1974). *Structural complexity of proofs*. Ph.D. thesis, Stanford University.

Wadsworth, C. P. (1971). *Types and programming languages*. Ph.D. 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.