Visualizing the Turing Tarpit

Jason Hemann Eric Holk
Indiana University
{jhemann,eholk}@cs.indiana.edu

Abstract

Minimalist programming languages like Jot have limited interest outside of the community of languages enthusiasts. This is a shame, as the simplicity of these languages ascribes to them an inherent beauty and provides deep insight into the nature of computation. We present a fun way of visualizing the behavior of many Jot programs at once, providing interesting images and also hinting at somewhat non-obvious relationships between programs. In the same way that research into fractals yielded new mathematical insights, visualizations such as those presented here could yield new insights into the structure and nature of computation.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic*]: Lambda calculus and related systems; D.2.3 [*Coding Tools and Techniques*]: Pretty printers

General Terms language, visualization

Keywords Jot, reduction, tarpit, Iota

1. Introduction

A Turing tarpit is a programming language that is Turing complete, but so bereft of features that it is impractical for use in actual programming tasks, often even for those considered trivial in more conventional languages. Alan Perlis, in his *Epigrams on Programming* [14], exhorts the reader to "Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy." Perhaps, however, this warning need not apply to programming-language enthusiasts or connoisseurs of art. A well-designed minimalist programming language can exhibit a beauty of its own – an elegance borne of its simplicity. Jot, a language devised by Chris Barker and which forms the basis of our present work, meets that standard. Further, its streamlined structure and strong connection to fundamental aspects of programming language theory allow us generate artifacts from Jot programs, such as is shown in Figure 1. These artifacts are both interesting and meaningful, as through them we can gain insight into the behavior of and relationship between programs.

In this paper we begin by refreshing some of the theory behind this work, including the λ-calculus and its various reduction strategies, universal combinators and their application to languages like Iota and Jot (Section 2). Using this, we devise several strategies for plotting the behavior of large numbers of Jot programs (Section 3). We make several observations about the resulting visualizations (Section 4), describe related work in the field (Section 5), and suggest possibilities for future work (Section 6).

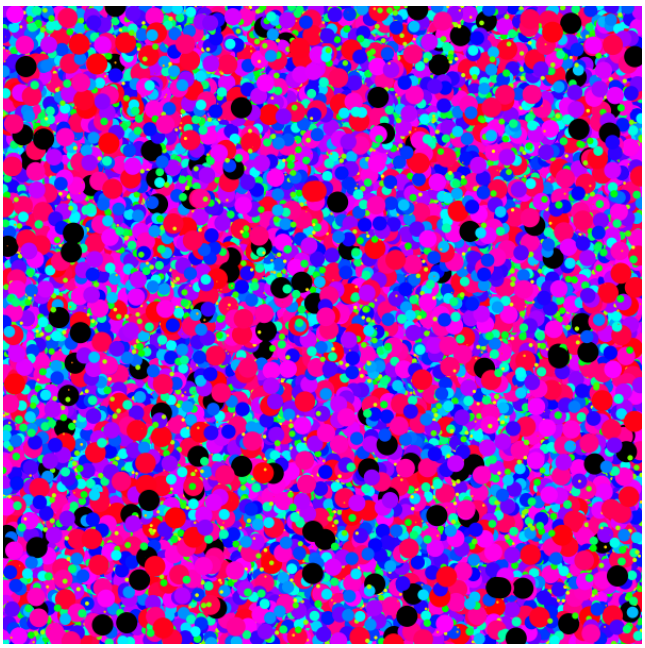

Figure 1. An example visualization of Jot programs plotted on a Hilbert Curve (Section 4.1). Each circle represents a single program. The color represents how long the program took to evaluate. Red took the longest, then blue, then green. Black circles represent programs that did not terminate within the allotted 10,000 evaluation steps. The radius of each circle is also derived from how long the program took to evaluate.

2. Background

We assume the reader is familiar with both the pure untyped λ-calculus (hereafter simply the λ-calculus) and combinatory logic. We briefly recapitulate a few known results from the literature essential to our exposition. Those seeking a more complete treatment should consult Hindley and Seldin [9] or Stenlund [21].

A *β-redex* (hereafter simply *redex*) is a λ-calculus term of the form $(\lambda x.M)N$. The term $[N/x]M$, which denotes the capture-avoiding substitution of N for x in M, is called the *contractum* of the redex. If a term T contains some redex we then replace by its contractum and call the resulting term T', we say the first term *β-contracts* to the second. If by a finite sequence of zero or more β-contractions a term T can be transformed to a term T', we say T *β-reduces* to T'.

Not every term contains a redex, and thus not every term β-reduces to a term other than itself. We say such terms are in *β-normal form* (usually, simply *normal form*), and the uniqueness of normal-forms up to α-equivalence (the Church-Rosser theorem) is a well-known result [2]. Recall that two terms are α-equivalent if one term can be transformed into the other only by renaming variables. Several less restrictive versions of normal form have been given attention in the literature, three of which we describe below. The relationship between all four normal forms is given in Table 1, reproduced from [18]. The metavariable E denotes a term in the relevant normal form, whereas e denotes an arbitrary λ-calculus expression. Each normal form in effect defines what reductions are legal, thereby leading to different behavior depending on the target normal form.

	Reduce under abstractions	
	Yes	No
Reduce args		
Yes	Normal form $E \to \lambda x.E \mid xE_1 \dots E_n$ **ao**, *no*, *ha*, *hn*	Weak normal form $E \to \lambda x.e \mid xE_1 \dots E_n$ **bv**
No	Head normal form $E \to \lambda x.E \mid xe_1 \dots e_n$ **he**	Weak head normal form $E \to \lambda x.e \mid xe_1 \dots e_n$ **bn**

Table 1. Comparison of four different normal forms and reduction strategies. Reproduced from [18]

These normal forms define the structure of a given term in normal form, they don't describe a method of bringing an arbitrary term *to* that normal form, assuming that term is, in fact, normalizable. In general, a given λ-calculus expression (e.g. $(\lambda x.y)\,\Omega$, where Ω is a non-terminating term) may have more than one redex, and the reduction may or may not terminate depending on the choice of redex.

A *reduction strategy* is an algorithm for selecting which redex to reduce at each step. Reduction strategies typically have the goal of reducing an arbitrary term to a particular normal form, if such a reduction is possible. Table 1 also catalogs several different reduction strategies. Applicative order (**ao**) reduction reduces the leftmost innermost redex first, and call-by-value (**bv**) reduces the leftmost innermost redex not inside a λ abstraction first. Call-by-name (**bn**) reduction reduces the leftmost outermost redex not inside a λ abstraction first,while head spine reduction (**he**) reduces the leftmost innermost redex not inside a λ abstraction (that is, redexes in head position). One can verify that each of these strategies do indeed produce only expression the relevant normal form. Three other reduction strategies are mentioned – these are fusion strategies mixing applicative order reduction with one of the other three. Each can be seen as a version of applicative order reduction in which, in the reduction of an application, the reduction of an operator is initially performed under another strategy. Normal order reduction (*no*), hybrid applicative order reduction (*ha*), and hybrid normal order reduction (*hn*) are the hybrid forms corresponding to the **bv**, **bn**, and *he* strategies, respectively. Interested readers should consult [18], which formed the basis of this discussion, for an extremely lucid treatment of the foregoing material.

2.1 Combinatory Logic

A *combinator* is a λ-term that contains no free variables. Combinators are useful for combining and transforming other fragments of code. It turns out that it is possible to define a set of combinators that are complete–that is, any computable function can be expressed in terms of these combinators. A set of combinators that is complete is called a *basis*. One such basis is $\{S, K\}$, where $S = \lambda x.\lambda y.\lambda z.xz(yz)$ and $K = \lambda x.\lambda y.x$, which serves as the foundation of the languages Iota and Jot.

2.1.1 X Combinator

Another result widely known from the literature is the existence of bases smaller than $\{S, K\}$, that is, bases of a single combinator [2]. The term $\lambda f.fSK$ is a universal combinator; as it has no free variables, it is indeed a combinator, and S and K are each recoverable from a series of applications of it. Letting X be the above combinator, S and K are recoverable via the applications of X below.

$$X(X(X(XX))) \Rightarrow S$$
$$X(X(XX)) \Rightarrow K$$

2.2 Iota

Iota is a language described in [3], based on the universal combinator given above. Its syntax and semantics are both simple and straightforward. For ease of exposition and a symmetry with the following section, we use the syntax for Iota given in [20] rather than that of the original. The valid characters in an Iota program are `0` and `1` (`i` and `*` in Barker's original description). The grammar below generates the valid Iota expressions.

$$I \to \texttt{0} \mid \texttt{1}\ I\ I$$

The symbol `0` denotes the universal combinator given above, and `1` is a prefix operator that syntactically denotes an application. As a consequence, there must always be exactly one more `0` than `1` in a well-formed Iota term, and the only legal Iota term not beginning with `1` is `0` itself.

2.3 Gödel Numbering

Gödel numbering is a technique for encoding programs as numbers. It was originally used to encode metamathematical statements in the formal system of *Principia Mathematica* as a part of Gödel's famous "Incompleteness Theorem" [12] For our purposes, it is useful as a way of converting programs into numbers that can then be used to plot properties about the execution of a given program.

2.4 Jot

Jot is a programming language related to Iota with the property that all binary strings are valid Jot programs [3]. It uses the same two characters as Iota, however the semantics are slightly different.

It operates similarly to Iota, but instead of being syntactic `1` is now an operator in the language itself. The duty of

Syntax			Semantics
$[\![F]\!]$	$\Rightarrow$	ϵ	$\lambda x.x$
	$\Rightarrow$	0	$[\![F]\!]$SK
	$\Rightarrow$	1	$\lambda xy.[\![F]\!](xy)$

Table 2. Syntax and Semantics of Jot, as defined in [3]

the X combinator have been split between `0` and `1`. The semantics are defined in terms of the λ calculus, and reference implementations use a call-by-value evaluation strategy.

The language features a straightforward translation from the SK-calculus to Jot (see Table 3 below). This translation is an injective map from the set of SK-calculus expressions to Jot expressions and has the property that every Jot program generated through it begins with a `1`, meaning that they are also unique binary numbers. This allows the translation into Jot to be used directly as a Gödel numbering of programs in the combinatory logic based on S and K ($\mathrm{CL_{SK}}$ expressions), and means that this subset of Jot programs are directly executable Gödel numbers.

$\mathrm{CL_{SK}}$		Jot
$[\![S]\!]$	$\Rightarrow$	`11111000`
$[\![K]\!]$	$\Rightarrow$	`11100`
$[\![AB]\!]$	$\Rightarrow$	`1`$[\![A]\!][\![B]\!]$

Table 3. Transformation of CL into Jot

Below, we exhibit a transformation of $\lambda f.\lambda x.f(fx)$, the Church representation of 1, into a Jot program. On line 1, we've used the standard SKI bracket-abstraction algorithm [4] to transform the expression into the SKI-calculus. A more sophisticated algorithms, such as that found in [23] would have resulted in a shorter SKI-calculus representation. On line 2, we replace all instances of I by *(SKK)*, transforming the original expression into an equivalent one in the SK-calculus. In line 3, we perform the transformation from the SK-calculus to Jot.

$$\lambda fx.f(fx) \Rightarrow S(S(KS)(S(KK)I))(KI) \quad (1)$$
$$\Rightarrow S(S(KS)(S(KK)(SKK)))(K(SKK)) \quad (2)$$
$$\Rightarrow \texttt{11111110001111111000111100111111000} \quad (3)$$
$$\texttt{11111110001111001110011111110000111001}$$
$$\texttt{11001111001111111000111001110}\texttt{0}$$

3. Implementation

In order to facilitate different sorts of experimentation, we implemented Jot in both Scheme and Javascript. We describe here interesting aspects of both implementations, and the full source code is available at `https://github.iu.edu/jhemann/jot-code/`.

3.1 Scheme

We first implemented a call-by-value reducer to weak normal form, and used this to implement the semantics of Jot. This implementation enabled us to read the expression that was the output of the reduction, which would otherwise have been represented merely as a closure.

We chose also to implement languages similar to Jot, but with slightly different semantics. As the meaning of Jot programs are defined in terms of the λ-calculus, different reduction strategies and normal forms generate different meanings for Jot programs. Using the six other reduction strategies to the four normal forms described in (Section 2), we created straightforward, naïve implementations in Scheme of reducers for each strategy to its normal form. Instead of binary strings, we take input in the form of lists of binary integers. We then augmented all of these implementations using the state monad to count the number of β steps taken as a part of each reduction.

Below, we demonstrate the most interesting case of β reduction: that in which perform an α substitution before doing the β, in order to avoid variable capture. We use a generated symbol (or `gensym`), as that is guaranteed to be a fresh variable in our expression. In order to handle programs which loop infinitely, and as will be described in Section 4, to provide an upper limit on the possible number of β reductions for our visualization, we limit here the possible number of β steps to $2^{8*3}-1$, which is the largest available value for a 24bit RGB value.

```
(define beta
  (lambda (M x e)
    (pmatch e
      ...
      ((lambda (,y) ,body) (guard (not (eq? x y))
                                  (free? x body)
                                  (free? y M))
       (let ((g (gensym)))
         (do (s1 <- get)
             (if (< s1 MAX_BETA)
                 (do (put (add1 s1))
                     (r1 <- (beta g y body))
                     (s2 <- get)
                     (if (< s2 MAX_BETA)
                         (do (put (add1 s2))
                             (r2
                               <- (beta M x r1))
                             (return
                               `(lambda (,g)
                                  ,r2)))
                         (do (put MAX_BETA)
                             (return '_))))
                 (do (put MAX_BETA)
                     (return '_))))))
      ...)))
```

Figure 2. The interesting case of `beta`. We perform an α substitution before doing the β, in order to avoid variable capture.

In Figure 3, we provide the implementation of the normal order reduction to normal form. The `pmatch` macro is a simple pattern-matcher used merely for cleanliness of implementation. The procedure `bv` is the actual call-by-value, weak normal form reducer, and `S` and `K` are the S and K combinators. The procedure `jot-interface` is provided merely to simplify invocation; it takes an initial string and provides the identity function as the initial procedure and 0 as the number of β reductions.

3.2 JavaScript Evaluator

We created an alternate implementation in JavaScript that facilitates experimentation in the web browser. As a first step, we created straightforward translations of the S and K combinators into JavaScript and then created a function that translates binary strings into JavaScript expressions that the browser's JavaScript engine can then execute.

In order to handle nonterminating programs and conveniently count the number of steps in the programs execu-

```
(define jot-bv-wnf
  (lambda (bls v)
    (pmatch bls
      (() (return v))
      ((1 . ,dbls)
       (jot-bv-wnf dbls `(lambda (x)
                           (lambda (y)
                             (,v (x y))))))
      ((0 . ,dbls)
       (do (n-v <- (bv-wnf `((,v ,S) ,K)))
           (jot-bv-wnf dbls n-v))))))

(define jot-bv-wnf-interface
  (lambda (bls)
    ((jot-bv-wnf bls '(lambda (x) x)) 0)))
```

Figure 3. Scheme implementation of call-by-value reduction to weak normal form.

tion, we converted these combinators to continuation passing style (CPS) and trampolined them [6]. Then, the driver function is able to count execution steps and also cut off programs that take too long to execute.

4. Visualizing Programs

The Jot language makes an excellent target for program visualization techniques. Due to its simplicity, it is easy to implement, and thus easy to experiment with changes to the implementation. Further, the natural mapping of the SK-calculus into Jot means it is easy to use to investigate well-known phenomena from the λ-calculus as they manifest in Jot.

Though we could have chosen to use any of a number of metrics by which to measure Jot programs, we chose to use the number of β steps required in the execution of the program as our metric. In Section 5, we suggest other possible metrics.

Using the Scheme implementation, we investigated the impact of the reduction strategy and normal form on the evaluation of Jot programs. As the transformation from $\mathrm{CL}_{\mathrm{SK}}$ to Jot (illustrated in in Table 3) shows, this trivially denumerable sequence contains within it the Gödelized encodings of all the expressions in the SK-calculus. We chose specific initial subsequences of the binary numbers, treated as Jot programs, as our data set.

We evaluated the Jot programs corresponding to roughly the first million (1024×1024) integers under each of the different reduction strategies, calculating the number of β reductions required for each program to terminate under each reduction strategy to its normal form. We mapped the number of β reductions to a 24-bit RGB value, and mapped each program into a distinct pixel in a 1024×1024 pixel image beginning in the upper left-hand corner and proceeding by row. Programs not terminating in under $2^{8*3} - 1$ β steps are assigned that number as their value, as this is the largest expressible integer using 24-bit colors. The corresponding pixels appear white in the resulting images. We also produced 2048×2048 and 8192×8192 pixel images using the same mapping technique.

One striking result is the self-similarity that appeared. This is most readily seen in the by-value reduction to weak normal form (Figure 4) and the by-name reduction to weak head normal form. It is interesting that this effect is more visible when reducing to normal forms that do not require

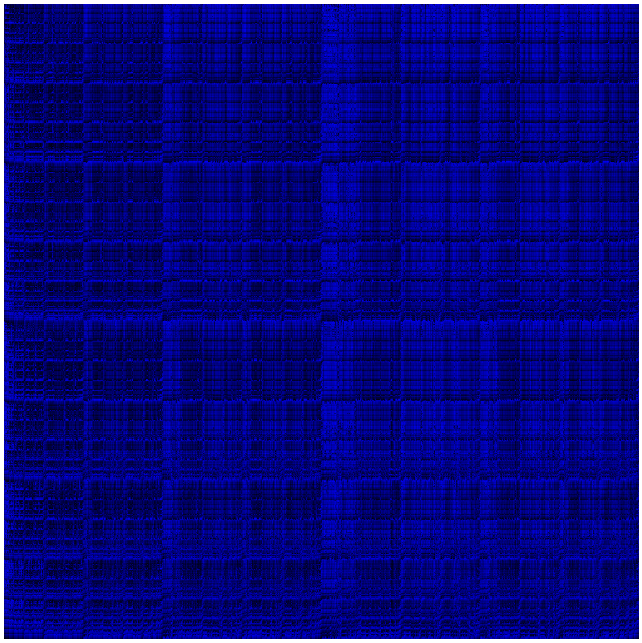

Figure 4. β steps in the call-by-value reduction of the first 1048576 binary integers, taken as Jot programs, to weak normal form

reducing under λ-abstractions. Similar structure is visible for reduction strategies that do, though, with larger images.

A second interesting result is the similarity in the images of the different normal form reduction strategies. The images for the four reductions to normal form – applicative order, and the three hybrid reductions – look strikingly similar. The significant differences between them are brighter spots in the images for the hybrid reduction schemata in areas where the applicative order reduction image was black. This indicates that these hybrid reductions took more steps to reach a normal form than did the applicative order reduction for these programs. Zooming in closer to the images reveals white pixels present in the applicative order reduction image that are not present in the hybrid ones. All of those we hand inspected were in fact programs which loop infinitely under applicative order reduction.

Thirdly, many of the programs which failed to terminate within the maximum allowed steps were within regular distances from one another. In several places, such as Figure 7, a close-up from the lower-middle section of Figure 6, there exist small clusters of computations that do not halt within the required number of β steps. Examining the non-terminating programs themselves make clear why this occurs. Below are the first five such programs from Figure 5.

```
  (1 0 0 0 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0)
  (1 0 0 0 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0)
  (1 0 0 0 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0 1)
(1 1 0 0 0 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0)
  (1 0 0 0 1 1 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0)
```

These programs all exhibit a common subsequence, which is the first program itself. The first of those programs reduces under applicative order reduction to the following expression.

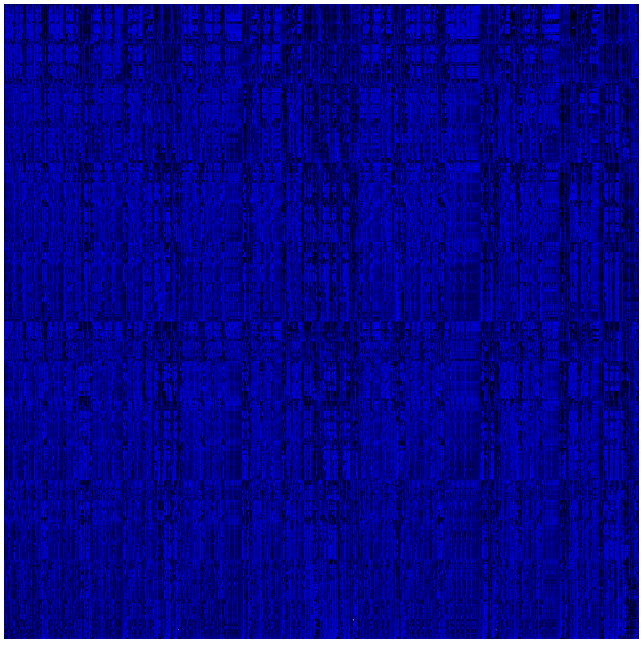

Figure 5. β steps in the normal order reduction of the first 1048576 binary integers, taken as Jot programs, to normal form. The applicative-order version looks remarkably similar.

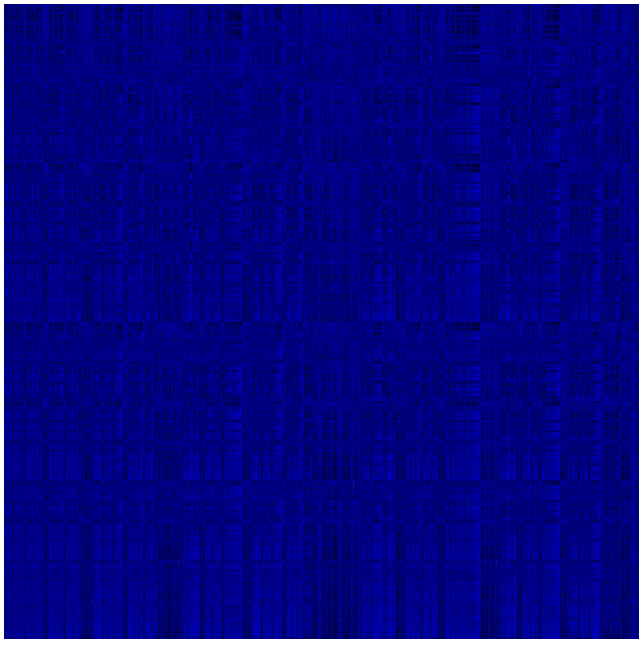

Figure 6. β steps in the applicative order reduction of the first 67 million binary integers, taken as Jot programs, to normal form

```
(lambda (x)
  (((lambda (y)
      (lambda (z)
        (lambda (x) ((z x) z))))
    x)
   (lambda (y)
     (lambda (z)
       (lambda (x) ((z x) z))))))
```

The applicative-order reduction of this term will fail to terminate.

Also surprising is how quickly all of the terminating programs did so. Very few of the programs we ran that terminated did so in more than several hundred β steps. As a result, our images contained little data in the red or green ranges. This appears to be an artifact of the size of our datasets – the largest program used in our visualizations here was only 26 bits in length. A larger and more diverse set of Jot programs will demonstrate other interesting characteristics when visualized.

4.1 Hilbert Curve Embedding

We present an alternate visualization, a mapping of Jot programs onto the Hilbert Curve, a well-known space filling fractal [8]. The Hilbert Curve may be used to map one dimensional data into two dimensions, such as in Randall Munroe's "Map of the Internet" [11], which provided inspiration for the visualization we present here. One particularly useful property of the Hilbert curve is that it tends to map points that are nearby in one dimensional space to points that are nearby in two dimensional space.

At first it seems a straightforward reinterpretation of Jot programs as binary numbers would suffice for mapping Jot programs into two dimensional space. Unfortunately, the sequences such as 1, 01, 001, … all represent the same binary number yet different Jot programs. Instead, we chose to

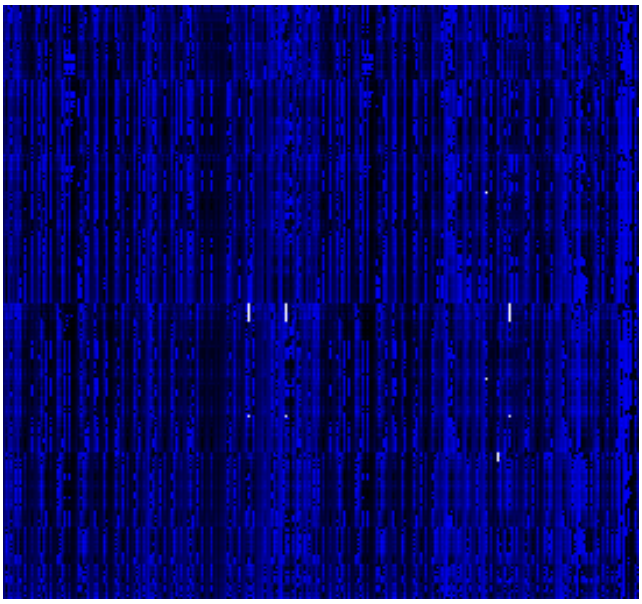

Figure 7. A Cluster of programs which did not terminate within the alloted number of βs. Several have been shown by inspection to be infinite loops.

view Jot programs as something like an address for a binary search through the interval $(0, 1)$. Thus, if we have a Jot program represented as a sequence of bits $x_1x_2x_3 \ldots x_n$, we can compute the real number representing this program as follows.

$$\frac{1}{2} + \sum_{i=1}^{n} \frac{(-1)^{1-x_i}}{2^(i+1)} \tag{4}$$

According to this equation, the empty string maps to 0.5, 1 maps to 0.75, 0 maps to 0.25, 11 maps to 0.875, etc. This mapping has the property that programs sharing a common prefix are mapped to nearby locations on the interval $(0, 1)$. We suspect programs with a common prefix are likely to have similar behavior, so this closeness property is desirable. Once we have mapped a program onto a real number, we then map this into two dimensional space using a Hilbert mapping and use the resulting mapping to generate images such as the one in Figure 1. As predicted we do indeed get striking patterns from this larger data set.

5. Related Work

Program visualization has long been an integral part of the software development process, including use in design, debugging, data-flow analysis, and performance profiling among others. A number of surveys [5, 13, 15] on the subject are available in the literature, and provide background and history of the program visualization generally. Though the majority of these are focused on imperative programming, [24] specifically focuses on program visualization in functional languages. All are much more concerned with program visualization from a software engineering perspective, and focus on much larger languages.

A number of tools to visualize aspects of the λ-calculus have been created; here we describe only a few. In [10], the author focuses on graphical representations of λ terms themselves. The work of [22] and [16] provide tree representations of the terms themselves and interactive reductions, whereas [7] provide more stylized graphs to represent the reductions. The system in [17] demonstrates exactly the reductions and normal forms we use in the present work. In [1], the author provides a visual method for representing combinatory logic akin to its development in [19].

6. Conclusions and Future Work

With these methods of mapping Jot programs into 2-D space, we have the framework to investigate relationships between various sets of Jot programs. It would be interesting to investigate the impact transformations on the original data set (e.g. inverting inverting the binary strings or flipping their bits) would have on the image. Generally, finding better selection criteria for the programs we choose to visualize might make for a more meaningful data set and more striking images. Further, there is no reason to be limited to visualizing programs based solely on the number of β steps taken. We could easily have chosen to instead represent the size of the resultant reduced term, or given different weights for the distinct lines of β substitution.

Different embeddings into 2-D space might also yield interesting results. For instance, plotting the programs into a circle around the origin, in which each ring would represent programs of length r, where r is the distance from the origin. Finally, changing the way in which we mapped the number of β steps to hues might yield more visually appealing images.

Much significant work has been done in visualizating programs, but always using more complex languages than Jot. Perhaps visualization of such a minimal language will enable, under continuing experiments, a better understanding of methods of program visualization itself, abstracted away from the specifics of particular language features. The visualization of such minimalist languages as Jot may yield new insights into the structure and nature of computation. To see these and other images at a higher resolution and watch our image generator in action, readers are invited to view our gallery at `http://eholk.github.io/TarpitGazer`.

References

[1] To dissect a mockingbird.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103. North Holland, 1985.

[3] C. Barker. Iota and jot: the simplest languages?, June 2013. URL `http://semarch.linguistics.fas.nyu.edu/barker/Iota/`.

[4] H. B. Curry and R. Feys. Combinatory logic, volume i of studies in logic and the foundations of mathematics, 1958.

[5] S. Ellershaw and M. J. Oudshoorn. *Program visualization-the state of the art*. Citeseer, 1994.

[6] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Proc. 4th ACM SIGPLAN international conference on Functional programming*, pages 18–27, New York, NY, USA, 1999. ACM.

[7] N. B. B. Grathwohl, J. Ketema, J. D. Pallesen, and J. G. Simonsen. In *Proc. 22nd Intl. RTA*, volume 10, May .

[8] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:469–460, 1891.

[9] J. R. Hindley and J. P. Seldin. *Lambda-calculus and combinators: an introduction*, volume 13. Cambridge University Press Cambridge, 2008.

[10] V. Massalõgin. Visual lambda calculus. Master's thesis, Rijksuniversiteit Groningen, Estonia, 2008.

[11] R. Munroe. Map of the internet. `http://xkcd.com/195/`, December 2006.

[12] E. Nagel. *Gödel's proof*. New York University Press, New York, 2001.

[13] M. Oudshoorn, H. Widjaja, and S. Ellershaw. Aspects and taxonomy of program visualisation. *Software Visualisation*, 7:3–26, 1996.

[14] A. J. Perlis. Epigrams on programming. *SIgPLAN Notices*, 17(9):7–13, 1982.

[15] G.-C. Roman and K. C. Cox. Program visualization: The art of mapping programs to pictures. In *Proc. 14th international conference on Software engineering*, pages 412–420. ACM, 1992.

[16] D. Ruiz and M. Villaret. Tilc: The interactive lambda-calculus tracer. *Electronic Notes in Theoretical Computer Science*, 248(0):173 – 183, 2009.

[17] P. Sestoft. Standard ml on the web server: Visualizing lambda calculus reduction. Technical report, Citeseer, 1996.

[18] P. Sestoft. Demonstrating lambda calculus reduction. In *The essence of computation*, pages 420–435. Springer, 2002.

[19] R. M. Smullyan. *To Mock a Mockingbird: And Other Puzzles*. Oxford University Press, 1985.

[20] M. Stay. Very simple chaitin machines for concrete ait. *Fundamenta Informaticae*, 68(3):231–247, 2005.

[21] S. Stenlund. *Combinators, λ-Terms and Proof Theory*. D. Reidel, Dordrecht, 1972.

[22] M. Thyer. Lambda animator, June 2013. URL `http://thyer.name/lambda-animator/`.

[23] J. Tromp. Binary lambda calculus and combinatory logic. *Kolmogorov Complexity and Applications*, 6051, 2007.

[24] J. Urquiza-Fuentes and J. Á. Velázquez-Iturbide. A survey of program visualizations for the functional paradigm. In *Proc. 3rd Program Visualization Workshop*, pages 2–9, UK, july 2004.