ICFP: U: Warm Fuzzy Things
A Domain Specific Language for Hypertexture

Samantha Frohlich

1 Problem and Motivation

The field of Programing Languages (PL) has much to offer the wider community. There are so many things that would benefit greatly from being formalised into a language. Sometimes, a simple library just doesn't cut it. By creating a language, the door is opened to greater expressively, reliable correctness, improved usability, and the ability to analyse multiple aspects of what is described. What's more, if the language is made domain specific, that it, it is specialised to not describe everything, but just one particular thing, the perfect median for discourse between experts can be found. By customising a language to a domain, the level of abstraction can be tailored such that irreverent details can be ignored, giving experts of that domain the opportunity to get straight to the heart of what they want to say, unhindered. In the same way that programming FPGAs to do a particular task can lead to great efficiency gains [2]; designing a language to match its domain can accelerate creation within that domain. The future is domain specific, and PL can lead the way.

This call for programming linguists to apply themselves in other fields and share their knowledge is not new. In a talk at ICFP 2018 [6], Pat Hanrahan implored those within PL to look a little further from home. He explained how again and again in his career there was a need for a language and he, being the only one there, had to fill it. At Pixar he developed the RenderMan Shading language, which is still used today. Now Hanrahan himself confesses that he is no PL expert, imagine what could be achieved if the people that really knew PL got involved. A similar story is told by Dan Piponi regarding how he created a Domain Specific Language (DSL) embedded into Python for describing blob like 3D shapes or "blobbies" [12].

It is about time that this call is answered. This work presents a formal semantics and embedded Domain Specific Language (eDSL) for hypertexture [11]. Hypertexture is a method for representing exciting three dimensional shapes, such as fire or smoke. It is an excellent candidate for formalisation because of three reasons: it is an interesting domain, its design lends itself perfectly to becoming a language, and it would benefit greatly from the application of PL techniques.

Nothing is more exciting than the glitz and glam of the Oscars, and that is exactly what hypertexture brings to the party. Having helped Perlin win an Academy Award, it is widely used in the film industry. This movie star has featured in films such as Disney Pixar's Ratatouille [1], where it played the staring role of the realistic blooming of bread.

Upon reading the paper that introduces hypertexture, it seems that Perlin too was pioneer of using a language to describe something. The paper presents functions that are clearly primitives, functions for combining hypertextures, and exciting results that can be achieved through use of these functions. However, since the paper predates advancements made in the last 30 years of PL, Perlin simply lacked the know-how to bring all these pieces together into a language. This leaves room for improvement.

While the phenomena Perlin introduced were beautiful, they were difficult to create. There was also nothing ensuring that the programmer of a hypertexture kept the density values between 0 and 1. With a language these problems of low usability and lack of correctness guarantees are solved. Additionally, a language allows a variety hypertextures to be expressed with ease. Instead of editing bespoke code to change one, the expression describing the hypertexture can simply be changed. This work can be seen as a revamp of the original paper, bring it bang up to date by sprinkling a little bit of PL magic.

2 Background and Related Work

2.1 Hypertexture

Normally, a three dimensional shape is represented digitally as a shell, where points in three dimensional space are either inside or outside the shell. When three dimensional shapes are represented as a hypertexture, it is as an array of density values. Each point in three dimensional space is assigned a density value in the range [0,1], where 0 is not part of the shape, and 1 is completely solid. Intuitively, smoke is a hypertexture as it is a billowing collection of points with low density values. Combining this representation with Perlin noise [9][10][8] can create beautiful and natural hypertexture phenomena like fur or eroded stone.

2.2 Type Class Morphisms

What As a method for designing semantics of a language, type class morphisms [3] are the marriage of Haskell type classes and denotational semantics. The Haskell type class part allows the specification of the desired operations; the denotational semantics part provides extra guidance of how each operation should behave.

Recipe This design process has two main steps:

1. Decide upon a mathematical model that represents the domain.
2. Add operations to the language. The semantics of each operation should be expressed denotationally and have the meaning of that operation in the underlying mathematical model.

Example To design the semantics of a language that creates key to value maps, the mathematical model could be a partial function from keys to values. Utilising a partial function instead of a total function allows the language to deal gracefully with the case that a given key is not in the map. Such a model can be expressed as follows:

$$[\![\ Map\ k\ v \ ]\!] \ = \ k \ \rightarrow \ Maybe\ v$$

Here the mathematical model is expressed using semantic brackets, the type of the language is on the left of the equals, and its meaning is on the right.

If the operations *empty* and *insert* were to be added to this language, their semantics would have to be that of *empty* and *insert* on partial functions. The *empty* partial function has no defined results implying that the semantics for *empty* in key-value maps should be *const Nothing* (no matter what key is given, *Nothing* is always returned). When it comes to *insert*, there are arguments to play with: the key and value to be inserted, and the map that will become their new home. Again, to discover the semantics for *insert* in maps, the meaning of *insert* for partial functions should be considered. How is a partial function expanded to accept another input? Well, an extra case is added to the function. Therefore, the map should be extended such that when it is asked for a value, it also checks if the given key matches the added key, and if so, the map should return the added value, if not, the original map should be consulted.

The semantics of operations are expressed as a semantic function:

$$\begin{aligned} [\![.]\!] &:: Map\ k\ v\ \rightarrow (k\ \rightarrow\ Maybe\ v) \\ [\![empty]\!] &= \lambda k \rightarrow Nothing \\ [\![insert\ k'\ v\ m]\!] &= \lambda k \rightarrow if\ (k\ ==\ k')\ then\ (Just\ v)\ else\ ([\![m]\!]\ k) \end{aligned}$$

The above function has a case for each operation mapping the keyword and its arguments to their semantics in the chosen mathematical model.

Benefits There are two main benefits to using type class morphisms. Firstly, the desired behaviour of each function is unambiguous. Secondly, the mathematical basis provides certain guarantees. If the underlying mathematical model has properties or laws, such as some operation being commutative, and the language faithfully represents this model, then these laws and properties will also hold in the language. i.e. that commutative operation from the mathematical model will also be commutative in the language.

2.3 Other Languages

There are other languages out there for computer graphics, such as the afore mentioned RenderMan shading language and Piponi's Python eDSL. The PL community has produced ReIncarnate [7], which is relevant as it also functionally describes three dimensional shapes. While this work also features affine transformations and boolean operations over shapes, they focus on how a shape is decomposed and cannot express anything like the hypertexture phenomena. Another graphics language from PL is Diagrams[1] [15], which, like this work, is a DSL embedded into Haskell. Unlike hypertexture, it (currently) sticks to two physical dimensions. (Note that it does have the ability to create animations with the third dimension of time.)

2.4 Uniqueness of Approach

When it comes to other work on hypertexture, no one else has turned it into a language. Previous work has looked into how the creation of hypertexture is challenging [5], and how it can be applied to more interesting objects such as CT scans [13]. However, hypertexture has never been approached from a PL perspective. Producing a formalisation of hypertexture not only makes them easier to create, by providing a creation tool in the form of a language, but also allows for further analysis and insurance of correctness.

[1]https://diagrams.github.io/

3 Contributions and Results

3.1 Formal Semantics

Mathematical Model The mathematical model for hypertexture went through a couple of iterations. Initially, it was the most obvious thing: a function from point to density. This is taken straight from the original paper [11], where, when specifying the boolean operations over hypertextures, Perlin models them as such. The model was clear and to the point, however, it was much to general. For example, nothing about functions ensures that the density value of a point lies between 0 and 1. To personally ensure that such properties are upheld is quite the undertaking. At this point, the type class morphisms method should be taken advantage of to gain the desired correctness guarantees. If a mathematical model can be found that keeps densities between 0 and 1, then using that model as the basis for the semantics will give the language this property for free. Surely, there is something in mathematics that matches more closely to hypertexture.

The solution is hidden in the original paper: in an aside when defining the boolean operations, Perlin notes that fuzzy sets could also model hypertexture, he just falls short of taking advantage of this. Fuzzy sets are similar to normal sets, except that membership is not binary, but a continuous value in the range [0,1]. This is perfect. Fuzzy sets are a much better model than functions as they provide the desired guarantees: the membership value of a point can be its density, which keeps it within the desired range. Fuzzy sets are to normal sets, what hypertexture is to the traditional shell model of three dimensional shapes.

Operations Since the results of the original work are Oscar worthy, this language should be able to express every example hypertexture from the original paper. The goal of this work is to improve the way that these fantastic results can be achieved, without compromising on what kinds of results are possible. In the original paper the results were the star of the show, now it is the implementation's turn, with the results being a given. This means that the following are required: a way of making shapes, boolean operations, and a way of *modulating* these shapes. The following semantic function fulfils these requirements:

$$\begin{aligned}
[\![.]\!] &:: \mathit{Hypertexture} \rightarrow \mathit{Fuzzy\ Set} \\
[\![\mathit{odf}\ \mu]\!] &= \lambda p \rightarrow \mu\ p \\
[\![\mathit{union}\ \mu_a\ \mu_b]\!] &= \lambda p \rightarrow \mathit{max}\ ([\![\mu_a]\!]\ p)\ ([\![\mu_b]\!]\ p) \\
[\![\mathit{intersection}\ \mu_a\ \mu_b]\!] &= \lambda p \rightarrow \mathit{min}\ ([\![\mu_a]\!]\ p)\ ([\![\mu_b]\!]\ p) \\
[\![\mathit{complement}\ \mu]\!] &= \lambda p \rightarrow 1 - ([\![\mu]\!]\ p) \\
[\![\mathit{modulateDensity}\ f\ s]\!] &= f \circ [\![s]\!] \\
[\![\mathit{modulatePoint}\ f\ s]\!] &= [\![s]\!] \circ f \\
[\![\mathit{modulateGeometery}\ f\ s]\!] &= f[\![s]\!]
\end{aligned}$$

Here fuzzy sets are expressed as a *characteristic function*. As an alternative to denoting a fuzzy set by enumerating its elements paired with their membership value, this function maps an element to its membership[1] .
The first operation, *odf*, gives the user free reign over what shapes they want to apply hypertexture to by simply asking them to provide a characteristic function. All classic shapes can be converted to a characteristic function by taking the *implicit function*[2] that represents them, evaluating it at the given point and asking if the result is greater than or equal to zero. A result of less than zero indicates that the point is within the shape and can be given a membership of 1.

Additionally, soft objects (those with density values other than 0 or 1) such as the *softSphere* from the original paper, can be expressed as characteristic functions:

```
softSphere :: Double → Point → Point → Density
softSphere r centrePoint p
  | d < r       = (1 − d / r)
  | otherwise = 0
 where d = euclideanDist centrePoint p
```

This characteristic function, expressed in Haskell, defines a soft sphere at *centrePoint* with radius *r*. Soft spheres are completely dense at their centre and get less dense the further away from the center a point this, so this function calculates the density of a point proportionally to how far the candidate point is from the centre.

The next few operations, *union*, *intersection*, and *complement*, constitute the boolean operations of the language, and copy semantics from their sister functions in fuzzy sets.

Finally, there are the *modulate* functions. These adjust the density of points within the soft region of a shape, where densities are between 0 and 1. There are three classes of modulation that depend on how much information the modulation function needs. Density modulation functions have type (*Density* → *Density*) and simply adjust the resulting density. For example, making an object more dense, or adjusting how quickly the density of an object changes. In the fuzzy set semantics this is expressed as post-composition i.e. the density is calculated by the characteristic function, and then changed by the density modulation function. Point modulation functions, as

[1]The characteristic function is very similar to the initial mathematical model of point to density functions. The difference is that fuzzy sets have a different semantics for the boolean functions, which provides the fuzzy set laws.

[2]In maths it is normal to represent shapes as equations. Most people are familiar with *explicit functions*, where a variable is the subject. For example, at school, children are taught that the following represents a circle: $y^2 = 1 - x^2$. These functions can be made *implicit* by having no variable as the subject, meaning the implicit function of a circle is: $f(x, y) = x^2 + y^2 - 1$.

the name suggests, adjust points before they are given to the characteristic function, which is expressed as pre-composition. The effect is that a point takes the density of another. Most of the phenomena from the original paper can be created by using this type of modulation to add noise to the points. The most interesting class of modulation function, the one that allows the expression of hair, requires knowledge of the surrounding geometry to find out the density of a given point. To achieve this, the existing characteristic function is given as an argument to the modulation function, so that it can create a new one.

Despite doing everything required, there are still improvements that can be made to the language. As per Hanrahan and Piponi's request, the professionals have arrived and this language will have all the latest PL bells and whistles. Currently, the operations are much too powerful. Consider *odf*, this operation can implement everything the other operations can do. No language operation should be able to implement the whole language alone. This puts too much trust in the user of the language, and programmers should never be trusted - if there is a way to abuse/misuse a language they will do it. The modulate functions are also too powerful and should be hidden.

Removing power from the user is easy. It involves exchanging the powerful functions for applications of those functions that are approved by the language designer. For example, instead of giving the user *odf* and allowing them to create any shape they want, they can instead be provided with a list of primitive shapes to choose from.

Exchanging all powerful operations for approved applications leads to an explosion of operators, highlighting one final improvement that can be made to the semantics: modularity.

Final Semantics The final hypertexture semantics is modular, allowing for two things: the user to pick and choose what parts of the language they want, and the mathematical model can now be specialised to each module. The shape part of the language has no need to be a fuzzy set, it would make much more sense for shapes to be modelled as their implicit functions. Likewise, if the user didn't want to do anything hypertexturey and they just want to describe shapes, there is no need for a fuzzy set, here a normal set will suffice. Specialising the mathematical model to each module of the language has lead to the hypertexture language having three semantic layers (see Figure 1):

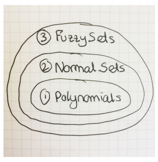

Figure 1: Layers of Mathematical Model

Polynomial (Implicit Function) Layer:

$$
\begin{aligned}
&[\![.]\!] :: \mathit{Shapes} \to f(x,y,z)\\
&[\![\mathit{cube}]\!] = \lambda x\ y\ z \to (\mathit{max}\ x\ y\ z) - 1\\
&[\![\mathit{sphere}]\!] = \lambda x\ y\ z \to x^2 + y^2 + z^2 - 1\\
&\ldots
\end{aligned}
$$

Normal Set Layer:

$$
\begin{aligned}
&[\![.]\!] :: \mathit{Boolean} \to \mathit{Set}\\
&[\![\mathit{primitive}\ p]\!] = \mathit{test} \circ p \text{ (Lifts shapes by evaluating and comparing to zero)}\\
&[\![\mathit{union}]\!] = \cup\\
&\ldots
\end{aligned}
$$

Fuzzy Set Layer:

$$
\begin{aligned}
&[\![.]\!] :: \mathit{SoftShapes} \to \mathit{Fuzzy\ Set}\\
&[\![\mathit{softSphere}]\!] = \mathit{softSphere}\ 1\ (0,0,0)
\end{aligned}
$$

$$
\begin{aligned}
&[\![.]\!] :: \mathit{GeomertyModulation} \to \mathit{Fuzzy\ Set}\\
&[\![\mathit{hair}\ f\ s]\!] = \mathit{hair}[\![s]\!]
\end{aligned}
$$

$$
\begin{aligned}
&[\![.]\!] :: \mathit{DensityModulation} \to \mathit{Fuzzy\ Set}\\
&[\![\mathit{constSoften}\ n\ ]\!] = (*n) \circ [\![s]\!]\\
&\ldots
\end{aligned}
$$

$$
\begin{aligned}
&[\![.]\!] :: \mathit{PointModulation} \to \mathit{Fuzzy\ Set}\\
&[\![\mathit{addNoise}\ ]\!] = [\![s]\!] \circ (\lambda(x,y,z) \to (\mathit{noise} * x, \mathit{noise} * y, \mathit{noise} * z)\\
&\ldots
\end{aligned}
$$

Unlike what is presented in the original paper, this linguistic method for creating hypertextures is expressive, usable, ensures correctness, and allows for analysis. Expressivity is gained from having a language to edit instead of specialised code, and the avoidance of such bespoke code also improves usability. Having a modular language further increases usability, while its mathematical basis ensures correctness (the fuzzy set laws ensure that the density values of composed hypertextures remain between 0 and 1). Finally, having a language that only allows correct hypertextures makes it easy for other semantics to be applied and other meanings explored. Clearly the application of PL techniques can enhance things greatly, even something that has won an oscar.

3.2 Implementation

The Haskell implementation contributed by this work is a *deep* and *modular* embedding. It is constructed using PL techniques from [4], and achieves the goal of being able to describe all hypertexture phenomena, including the star hypertexture: a fireball! (See figure 2.)

Deep There are two main types of embedding: shallow where the semantics are encapsulated as functions in the host language, and deep where operations are make up of the host's data types. Since the Hypertexture language is created as Haskell data type, it is the latter.

Semantics are applied to a deep embedding by a semantic function that recurses on the data type representing an Abstract Syntax Tree (AST). To make the language more amenable to analysis and earlier to understand, this recursion has been isolated using a data type called *Fix*:

$$\textbf{data}\ \mathit{Fix}\ f = \mathit{In}\ (f\ (\mathit{Fix}\ f))$$

The above allows the data types of the language to be recursion free, since the *In* constructor does the recursion for them. To tell *Fix* where to put the recursion, the recursion points in the data types are marked with a type parameter k (for continuation), then *Fix* provides *Fix f* as the type that should fill that hole, performing recursion in a safe and contained manner.

A catamorphism crushes the AST into its semantics:

$$\begin{aligned}&\mathit{cata} :: \mathit{Functor}\ f \Rightarrow (f\ b \rightarrow b) \rightarrow \mathit{Fix}\ f \rightarrow b\\ &\mathit{cata}\ \mathit{alg}\ (\mathit{In}\ x) = \mathit{alg} \circ \mathit{fmap}\ (\mathit{cata}\ \mathit{alg})\ \$\ x\end{aligned}$$

This function takes an *alg*, which knows how to unwrap one constructor of the AST into its semantics and recursively applies it to the fixed data type to yield the final semantic result. Changing semantics is as simple as changing the algebra, making it easy to have multiple semantics for an AST.

Modular Each module of the semantics is written as its own fixed data type. These modules can then be picked from by the user and combined using (: + :):

$$\textbf{data}\ (:+:)\ f\ g\ k = L\ (f\ k)\ |\ R\ (g\ k)$$

This combinator takes two syntactic functors and composes them together, allowing the user to choose from either f or g. Algebras can also be combined, pattern matching on the L and R constructors to decide which *alg* is required. To avoid the arduous task of writing out loads of *In*s and *L*s and *R*s, techniques from “Data Types à la Carte”[14] are applied to the hypertexture language.

Fireball The following Haskell code produces Figure 2:

$$\mathit{fireball} = \mathit{addTurbulence}\ \mathit{fireNoiseSettings}\ (\mathit{scale}\ s\ \mathit{softSphere})$$

addTurbulence is one of the point modulation primitives. It adds turbulent noise (repeated application of noise), configured by a settings data type, to a scaled soft sphere.

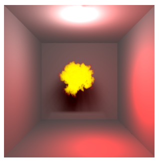
Figure 2: BOOM!

4 Future Work

Currently noise in this language is primitive that the user can adjust with settings. This prevents it from being analysed and limits the expressively of the language. The more that is internalised into the language the more that can be analysed. Rather than giving the user a predefined noise primitive, or shape for that matter, it would be better to provide them with the tools to make their own, allowing for analysis and even greater expressively. The next goal for the hypertexture language is to add an expression language that can describe different types of noise.

After that, who knows, there is a whole world of shapes to play with...(fractals anyone?)

References

[1] Session details: Course 6: Anyone can cook: Inside ratatouille's kitchen. In Apurva Shah, editor, *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[2] Mike Ashby, Christiaan Baaij, Peter Baldwin, Martijn Bastiaan, Oliver Bunting, Aiken Cairncross, Christopher Chalmers, Liz Corrigan, Sam Davis, Nathan van Doorn, Jon Fowler, Graham Hazel, Basile Henry, David Page, Jonny Shipton, and Shaun. Steenkamp. Exploiting unstructured sparsity on next-generation datacenter hardware. `https://myrtle.ai/wp-content/uploads/2019/06/IEEEformatMyrtle.ai_.21.06.19_b.pdf`, 2019.

[3] C. Elliott. Denotational design with type class morphisms (extended version). Technical report, 2016.

[4] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, 49, 08 2014.

[5] G. Gilet and J.M. Dischler. A framework for interactive hypertexture modelling. *Computer Graphics Forum*, 28(8):2229–2243, 2009.

[6] Pat Hanrahan. The role of functional programming and dsls in hardware. `http://blog.sigfpe.com/2007/11/blobby-language.html`, 2018. ICFP.

[7] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional programming for compiling and decompiling computer-aided design. *Proc. ACM Program. Lang.*, 2(ICFP):99:1–99:31, July 2018.

[8] K. Perlin. Improved noise reference implementation.

[9] K. Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[10] K. Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, July 2002.

[11] K. Perlin and E. M. Hoffert. Hypertexture. *SIGGRAPH Comput. Graph.*, 23(3):253–262, July 1989.

[12] Dan Piponi. A blobby language. `http://blog.sigfpe.com/2007/11/blobby-language.html`, 2007.

[13] Richard Satherley and Mark W. Jones. Hypertexturing complex volume objects. *The Visual Computer*, 18(4):226–235, Jun 2002.

[14] WOUTER SWIERSTRA. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[15] Brent A. Yorgey. Monoids: Theme and variations (functional pearl). In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, page 105–116, New York, NY, USA, 2012. Association for Computing Machinery.