Project Overview: Functional Programming and Data Visualization

This project demonstrates an embedded DSL for building interactive data visualizations with PureScript, using D3.js both as inspiration and as an implementation layer under the Foreign Function Interface (FFI).

The DSL uses a Finally Tagless approach that allows multiple implementations for the "language" which permits both alternative implementations (only one, D3, is currently shown) but also alternative interpreters which generate code, documentation or even further "meta" visualizations to allow one to manipulate the DSL syntax tree.

This overview document is an introduction to the ideas in the project and a guide to the other documentation and articles about it. Presuming that many people who might be interested in this work are going to come from only one of two disciplines: data-visualisation OR functional programming, i've tried to provide introductions for each camp to at least be acquainted with the important aspects of the other before reading further.

Motivation and goals

Motivation

I have built moderately complex, custom interactive data visualisations in the past both in JavaScript and PureScript, using D3.js. I found that JavaScript generally, and D3 in particular, seemed to work best for visualisations that were less "app-like" and more "chart-like". What i mean by this is that when the complexity started to rise to the level of a small application and when multiple programmers were involved, or if one had to return to some code after time had elapsed, the whole thing was very brittle and refactoring of it prohibitively difficult.

This could certainly be a "feature, not a bug" for some domains of application such as building a big beautiful rich visualisation for a one-off publication such as a New York Times feature. However, when the visualisation is used to *control* application behaviour or the visualisation begins to approach the complexity and multi-layered-ness of an app...this all in one single script language is a real problem, at least in my experience.

Other's experiences may vary -- some people have more tolerance and/or ability to handle those complexities. Nonetheless, most people who turn to languages like PureScript (Haskell family languages in general) do so because they share those experience and have found that there are better mechanisms for composing programs together which are more robust.

In PureScript it is common, and easy, to use JavaScript libraries via the FFI initially as it is a very quick way to get access to the enormous world of functionality that exists in open source JavaScript libraries. Sometimes this can be sufficient, you wrap a component or a function and its abstractions never leak and all is well. Other times, you wrap something but there's a kind of impedance mismatch with the way the JavaScript abstraction work and the way you'd like to handle, and particularly to compose, things in the purely functional world. D3.js was definitely the latter, for me.

D3.js is a big library with thousands of API end-points but, crucially, not all of those end-points are problematic for composing larger scale applications or weaving visualisations into PureScript web applications. Instead, its is primarily two core areas of the API, Selection and Simulation (more details on these later) which tend to actually *structure* programs in a characteristic D3 / JavaScript vernacular. It is these APIs that are first wrapped (by FFI) and then made available in purely functional idiomatic way by this library.

A secondary, but also very important, consideration is the ability to design and work with Algebraic Data Types (ADTs) and the rich container libraries that are available in PureScript while building and implementing visualisations and especially the code that surrounds the visualization. While D3 ultimately is a kind of array programming DSL *within* JavaScript and our PureScript eDSL is going to bottom out to some sort of "arrays mapped over the DOM" too, we want to be able to create data models that are more sophisticated and have better invariants as these are keys to both composability and maintainable, long-lived programs.

Goals

There are two "number one priority" goals: *expressability* and *readability*. Since it's impossible to have two number one priorities these are necessarily somewhat in tension and has been necessary at all times to try to balance them.

Expressability

Something that people with limited prior knowledge / experience of data visualisation often seem to find surprising is the degree to which D3 is *fundamentally different* from "a charting library". While the library has some affordances that make it very easy to do common visualisations it is not in any way about "canned visualisations". Rather, it is a language for describing a relationship between arrays of data and arbitrary constructions of DOM (HTML or SVG) element or marks on Canvas, and it could in principle be used to do auditory "visualisation" or, who knows, maybe olfactory "visualisation" or drone displays or whatever.

You can get a greater sense of the potential of D3 and the range of things that have thus far been produced using it at Observables and i will discuss this again in the section "Introduction to Data Visualisation for Functional Programmers.

So when we talk about expressability as a goal for this PureScript eDSL, we're not talking about working at the level of "make me a bar graph", "make me a scatterplot", we're talking about retaining the expressability of that translation from array to, for example, SVG.

Furthermore, I am especially interested in, and an advocate for, interactive data visualisations that act as control elements for other aspects of applications. It's worth calling this out here because "interactive visualisation" very often means "explorable" visualisation, ie one in which you can interact with the elements *but only to manipulate the visualisation itself*. That is a different, and more limited, sense of interaction from what i intend.

Before starting this project the question of criteria for "expressability" arose. My approach has been to analyse hundreds of existing visualisations that can be found on ObservableHQ and previously on `http://bl.ocks.org` and to select, model and then translate a set of examples that contained at least the most important structurally distinct visualisation types.

Things that the examples were chosen to show:

- simple use of selections and attributes
- use of the D3 design pattern known as the "General Update Pattern" together with transitions
- at least one example of `d3-hierarchy`, with six variations on the basic tree
- an example of the D3 simulation functionality to render graphs (ie nodes and edges) using a physics engine to do the layout

In addition, some of these examples feature other key ideas that i knew would be necessary to convince *me* that any new library was more than pre-canned charts. I'm referring to features such as dragging elements, panning and zooming the visualisation etc, here.

Finally, because these examples are all pretty small and don't really hit the limits of complexity that motivated this project i have also written a PureScript Halogen app that is more fully featured and designed to be interactive in that second sense, allowing actions in the visualisation to trigger Halogen level events.

Readability

If we look at D3.js as a kind of embedded DSL in JavaScript it is certainly clear and readable in it's core feature: declaratively associating some array(s) of data with some elements in the DOM and attributes of those elements. While it is definitely not a goal to reproduce the structures of D3js own language, it's also unwise to change it arbitrarily because there's a very large amount of real world experience to be leveraged or at least not shunned. So that's one kind of readability concern - can it be somewhat isomorphic to the equivalent JavaScript where that's appropriate?

A second form of readability is concerned with the places where - for me - the D3 / JavaScript approach breaks down, which is in roles and responsabilities of different parts of the code. This is related pretty closely to composability (see next goal for more on this).

Ideally, i would like the person coding the data layer and data model to be somewhat insulated from the concerns of the person using that data model to create a visualisation. And likewise, i would like the person developing the data visualisation to be somewhat insulated from the concerns of a web app developer. Now, these might very well all be the *same person* but separating the concerns like this makes it easier to evolve the code and, crucially, makes it all a little less brittle.

For example, PureScript's strong typing helps the compiler catch changes to the data model that would break the data visualisation at runtime and a capability-based API for the visualisation (see [Library Architecture](missing link) enables the web application developer to treat the visualisation as a component and the data model code as normal, idiomatic PureScript.

So "readability" here is multi-faceted and it can be summarized as:

- unless there's a good reason not to, preserve the declarative simplicity of D3's selection chaining and attribute setting
- tease apart the different roles and responsabilities in the code to disentangle data-model from visualisation from application framework code.

Composability

There is an additional important goal which is *composability*, if you are a PureScript or Haskell programmer you probably know what i mean by this and if you are, say, a JavaScript D3 programmer perhaps that will seem odd or even contentious. I will discuss composability at more length in the section "Functional Programming for Data Visualizers" below.

Non-goals

Non-goal: Complete API coverage

As alluded to above, there's lots of API in D3 that needs nothing more than an FFI wrapper to be accessible from a PureScript eDSL. D3 is both modular and somewhat functional in style (in the JavaScript sense of functional programming, to be clear). So it was from the start a non-goal to completely expose all of D3 as *idiomatic* PureScript where a simple wrapper was sufficient.

Furthermore, i have only written those wrappers *as I needed them* to there are still *many* parts of D3 that are not covered by this eDSL.

Non-goal: Modelling of D3 State

This might seem like a surprising choice - D3 is inherently *very* stateful, there's state in D3, there's state in the DOM, there's statefulness in your (pure) data after you give it to D3. State everywhere. In many cases in functional programming you might try to ameliorate the dangers of this by explicitly modelling the state, using a State Monad or marking everything that changes or depends upon state as "Effect"-full.

Indeed i have tried this approach in the past. In this library i have instead striven to isolate the statefulness to only the code that uses eDSL represented by the `Selection` and `Simulation` monads. This *significantly* removes but cannot fully eliminate the issues associated with state. However, i believe it is a good compromise although much more could be said on the matter.

Non-goal: performance optimisation that compromise readability

The performance bottlenecks in a D3 visualisation are, by their nature, going to be the assignment of attributes to (potentially millions of) DOM elements. It is likely that this code will always have to have a native implementation, ie FFI, *or* it would have to be re-written in a way that was as performant as possible in PureScript. I have chosen to use D3 for this as a low-level, well-tested, optimised layer in order to preserve readability of the visualisation code.

There definitely are other ways one could make this tradeoff, the approach taken has proved satisfactory thus far but other options would be very interesting to discuss and PRs are welcome, particularly if they are additive rather than replacements to what is here. A 100% PureScript alternative for D3's Selection, Attribute and Transition APIs would be very welcome even at a price of performance.

Data Visualization for Functional Programmers

> Computer-based visualization systems provide visual representations of datasets designed to help people carry out tasks more effectively. Visualization is suitable when there is a need to augment human capabilities rather than replace people with computational decision-making methods. The design space of possible vis idioms is huge, and includes the considerations of both how to create and how to interact with visual representations. Vis design is full of trade-offs, and most possibilities

> in the design space are ineffective for a particular task, so validating the effectiveness of a design is both necessary and difficult. Vis designers must take into account three very different kinds of resource limitations: those of computers, of humans, and of displays. Vis usage can be analyzed in terms of why the user needs it, what data is shown, and how the idiom is designed.
>
> *-- Munzner, Tamara. Visualization Analysis and Design (AK Peters Visualization Series) (p. 1). A K Peters/CRC Press.*

General overview: DataViz

Tufte, Minard, Anscombe etc

D3 particularly

Further reading

Tamara Munzner, FT guide, Wattenberger, Manuel Lima

Functional Programming for Data Visualizers

General overview: FP

Purity, totality, static types, type inference, category theory, composition, illegal states unrepresentable

---

Next steps: guide to other docs

Tutorials

Building the Three Little Circles example

Building the General Update Pattern example

Building the Trees example

Building the Les Miserables example

How-Tos

Building an App

Discussion

Finally Tagless Viz

How Finally Tagless encodings work and how they can be useful for eDSL's with multiple interpreters such as this library.

Roles and Responsabilities

Describes the particular approach to application development and the separation of concerns that is advocated / facilitated by this libary.

Swapping out, re-writing or augmenting D3.js

Possibility of using GraphViz for layout, of writing a PureScript native Selection monad etc etc

---

Reference

Library Architecture and API guide

(probably just Pursuit once published)

Layers of DSLs