Extending Legacy Software with Functional Programming

Christian Fischer

January 8, 2018

Contents

Chapter 1

Abstract

Chapter 2

Acknowledgements

Chapter 3

Introduction

> Legacy code. The phrase strikes disgust in the hearts of programmers. It conjures images of slogging through a murky swamp of tangled undergrowth with leaches beneath and stinging flies above. It conjures odors of murk, slime, stagnancy, and offal. Although our first joy of programming may have been intense, the misery of dealing with legacy code is often sufficient to extinguish that flame.
> — Robert C. Martin, Foreword to Working Effectively with Legacy Code [7]

Problems related to and caused by *legacy code* are ubiquitous. Maintenance of legacy systems costs billions of US dollars every year, and as time goes on, the number of legacy systems in production will continue growing, as what is considered modern today is likely to become "legacy" tomorrow. In fact, one does not need to go back far in time to find "legacy" code, as any code that is difficult to understand, and it is difficult to make changes to such that the effects of the changes on the system are predictable, can be considered "legacy" code[1].

Many solutions have been proposed and tried, yet legacy code continues to be a problem. It will continue to be a problem, and be solved not not when a way to "repair" legacy code is found, but when a way to write code that does not *become* legacy code is found[25].

This thesis is concerned with evaluating purely functional programming as one potential solution to and preventative measure against legacy code problems. This was done by developing a web application in PureScript (PS) that both embeds an existing JavaScript (JS) application, as well as provides novel functionality. The process, and resulting code, were evaluated by looking for signs of legacy code problems, as defined in the Method chapter.

3.1 Context

GeneNetwork 2 (GN2) is a web-based database and toolset for doing genetics online[16], developed at University of Tennessee Health Science Center. One feature yet to be implemented in GN2 is an interactive genome browser.

Biodalliance (BD) is one such genome browser, which is written in JavaScript (JS) and can be embedded in web pages[5]. However, BD is several years old,

and has accumulated a lot of code and features, which have led to difficulties when adding new functionality that is desired in GN2.

The Graph Genetics Browser (GGB) was developed as part of this thesis to solve these problems, by embedding BD and wrapping the desired features of BD in a new interface, as well as providing the groundwork for a new genome browser independent of BD. The development of GGB served to evaluate the hypothesis, described next.

3.2 Objective

The objective of this thesis is to evaluate the use of purely functional programming as a tool to:

1. Work with and extend an existing legacy system
2. Develop an application unlikely to develop problems associated with legacy systems

The legacy system in question was BD, the new application GGB. A BD browser is embedded in GGB, so that the functionality of BD is gained while providing a framework for a new browser without the legacy baggage of BD. As initial new functionality, the Cytoscape.js (Cy.js) graph network browser is also embedded, with support for some interaction between Cy.js and BD, managed by GGB.

The development of GGB provides an opportunity to evaluate the hypothesis that purely functional programming can assist when working with legacy systems, as well as lead to code that is less likely to exhibit legacy problems. This is done by identifying some general causes of legacy problems, and looking at the resulting codebase. Of course, this thesis concerns a single programming project and a single programmer; it can be seen as a case study.

3.3 Report Structure

This report begins with presenting, in the Theory chapter, the concept of legacy code, why and how it is a problem, and how the problems presented by legacy code have been attempted to be solved previously. The problems are distilled into heuristics for code quality, as tools to classify whether some code is or is not likely to exhibit legacy problems.

Those heuristics match well with code written in a purely functional style, which is what the second half of the chapter is concerned with. The purely functional programming (pure FP) paradigm is introduced, together with some characteristics and advantages of said paradigm.

The Method chapter begins by connecting the dots of legacy code and pure FP. The purely functional language PureScript (PS) is introduced, and some of the important characteristics of pure FP are explored in PS, together with their advantages as concerning the earlier defined heuristics.

Next, the process of the thesis project is presented. Biodalliance is introduced as a legacy system, and the Graph Genetics Browser as an application extending said system, as well as how this project fulfilled the thesis objective and provided the data necessary to evaluate the hypothesis.

The following Results chapter first presents the resulting browser, then goes into some detail on the implementation of various parts of the browser. These subsections serve to provide solid examples of how pure FP can solve problems associated with legacy code, and produce code that is less likely to exhibit similar problems in the future.

In the Discussion, it is found that GGB and its codebase meets the requirements. Then we go into more detail examining the suitability of PS for working with legacy JS, and which characteristics of BD made it more or less suitable for this process. Next a closer examination is made of what parts of pure FP helped with legacy problems, and the chapter ends with a walk through of some of the developmental difficulties encountered during the project.

The report concludes, having found that pure FP was a good tool especially for GGB and BD. Some other ways of gaining the advantages of pure FP, without using PS or another pure FP language, are given. This is followed by the limitations of the project, especially its nature as a case study, and some ideas for future studies. Lastly the future of the Graph Genetics Browser is presented.

With the hypothesis (pure FP as a viable tool for working with legacy code) and an overview of the project in hand, the next chapter dives into the concepts of legacy code and pure FP.

Chapter 4

Theory

The prerequisite theory for the thesis project is introduced. A definition of "legacy code" is given, followed by relevant statistics, facts, and techniques concerning legacy code and working with it. This is followed by an introduction to the purely functional programming paradigm.

4.1 Legacy Code

This section begins with defining legacy code, and presenting statistics concerning its prevalence and costs. Why it is a problem, some attempted solutions, and the causes in code of those problems, follow. The section ends with defining code quality with respect to legacy code.

4.1.1 Definition

There is no formal definition of "legacy code", but [7] gives the definition "Legacy code is code that we've gotten from someone else." Bennett provides a definition of legacy systems in general, which gives an idea of why it may be problematic: "large software systems that we don't know how to cope with but that are vital to our organization" [1]. Finally, Weide et al. gives a definition closer to the spirit of the concept as experienced by programmers working with legacy systems — in the trenches, as it were:

> [..] legacy code, i.e., programs in which too much has been invested just to throw away but which have proved to be obscure, mysterious, and brittle in the face of maintenance.

In other words, legacy code is code that continues to be relevant, e.g. by providing a service that is important, and that requires modification, or will require modification. If there were never going to be any reason to modify the code, it would not be worth talking about, nor is it likely that a system that provides a continually valuable service will not at some point in the future require maintenance or new features [11].

For this reason, legacy systems are prevalent in the world. If a system works as it should, providing the service that is needed, and said service must continue

to be provided, the safest thing to do is to leave it as is — until it is decided, for whatever reason, that changes must be made.

The U.S. goverment federal IT budget for 2017 was over $89 billion, with nearly 60% budgeted for operations and maintenance of agency systems, with a study by the U.S. Government Accountability Office finding that some federal agencies use systems that are up to 50 years old [17].

Many of these federal agency systems consist of old hardware, old operating systems, etc., however the problems of a legacy system do not need to be caused by such factors. The code itself is often the problem [2], and is what this thesis is concerned with.

We define a "legacy codebase" as the codebase of a legacy system, where the problem of modifying the system is constrained by the code itself — the underlying technology is not relevant. Likewise we do not look at dependencies, a problem solved by pure functional package managers such as Nix [4] and Guix [3].

Why would changes need to be made to a legacy codebase? When the behavior of the system needs to be changed. [7] identifies four general reasons:

1. Adding a feature
2. Fixing a bug
3. Improving the design
4. Optimizing resource usage

All of these somehow modify the behavior of the system; if there was no change in the system behavior, the change in code must have been to some part of the codebase that isn't used! Thus, the desired change requires a change in behavior. The problem with legacy code is that it is difficult to know how to make the change to the code that produces this desired change in behavior, and *only* the desired change.

The main reason it is difficult to work with legacy code is lack of knowledge of the system and codebase, and how the system's behavior relates to the underlying code. Legacy codebases often lack documentation and tests, without which a new programmer on the project has few, if any, tools at their disposal to understand the codebase as it is, since they do not have any knowledge of how and why the code came to be as it is. Even if a design or system specification exists, it is not necessarily accurate. The code may very well have grown beyond the initial specification, and the specification need not have been updated in step with the code.

For these reasons, one of the main problems of working with legacy code is understanding it in the first place [7] [1] [21]. This is also a difficult, time-consuming process, and one of the reasons reverse engineering legacy systems is rarely, if ever, a cost-effective undertaking [25]. Also according to Weide, Heym, and Hollingsworth, even if a system is successfully reverse engineered and modified, even if a *new* system is successfully developed that provides the same behavior as the legacy system but with a better design, it is highly likely that the new system, eventually, reaches a point where it too must be reverse engineered — i.e. when the "new" system becomes another legacy system, and the cycle begins anew.

In short, the problem with legacy code is lack of knowledge in what the system does, and how the code relates to the system and its parts. This makes it difficult to know what changes to make to the code to produce the desired change in system behavior, and if a change made is safe, i.e.\ that *no* undesired change in system behavior results.

4.1.2 Solutions

Legacy code has been a recognized problem for decades, and people have been trying to solve it for as long. [11] [25] [14] This section will present some general approaches and tools that have been applied when working with legacy systems. First, techniques that help with manual reverse-engineering, which is the most widely used approach when working with legacy code, are introduced, followed by a brief walk through some automated tools.

Reverse Engineering Legacy Systems

Reverse engineering a system can be very difficult and time-consuming, even with access to the source code [25]. There are entire books dedicated to the subject of understanding a legacy codebase and working with it to transform it into something more easily understood and extended [7].

As previously mentioned, the greatest problems stem from insufficient knowledge of the system; from not knowing what code does, or what happens when a change is made. Thus, adding tests is one of the main tools for increasing understanding, as tests provide the programmer with feedback on their code changes. [21]

However, this feedback is only as good as the test suite; if a system behavior is not covered by the tests, the programmer has a blind spot in that area.

Where and how to add tests is an art and science of its own, as the programmer must find what parts of the system need to be tested, and how to insert the tests into the codebase. This becomes more difficult when a program has many tightly-knit parts, global state, etc.

With a robust test suite, a programmer can modify the codebase to improve the code quality and architecture of the system, confident that their changes do not compromise the system's behavior.

Automated Approaches

Automated tools that assist with legacy systems are largely concerned with increasing the knowledge of the system, understanding how it works, extracting modules, etc. One example is extracting OOP abstractions such as classes from imperative code, e.g. by analyzing which functions and variables are used together. [6] [22]

Another interesting route is creating a "modularization proposal," i.e. a series of architectural changes to the codebase that lead to a more modular system while minimizing change each step, by constructing a concept lattice based on where different global variables are used. This lattice can then be used to create descriptions on how to modularize the codebase. [13]

Another approach is using automated tools to detect potentially problematic parts of a codebase, such as overly complex controllers in GUI applications [12],

or code that is simply more likely to be buggy, based on statistical analyses [19]. These would help programmers find what parts of the codebase to target.

Some ways that developers and researchers have attempted to fix legacy codebases have been introduced, however one question still remains: what is it with legacy code that makes it problematic? Is there some attribute of the code itself, or is *all* code that successfully solves a problem doomed to become "legacy" code? I.e., is it possible to write code that is "legacy"-resistent?

We have identified that the problems seem to be related to understanding the code. From that viewpoint, the question becomes: is there a way to write code that is more easily understandible, and if so, what characterizes it? This is what the next section seeks to answer.

4.1.3 Code quality

The problems of legacy code revolve around knowing what different parts of the codebase do; what changes in code lead to what changes in behavior, and what changes in code do not lead to changes in a subset of the system behavior. "Good" code could, then, be defined as code which:

1. Tells the programmer what it does
2. Tells the programmer what it does **not** do

Conversely, "bad" code makes it difficult to see why it exists, why it is called, and what effects it has, and does not have, on the system behavior.

These definitions only beg further questions, however, and require deeper investigation.

Knowing what a piece of code does means knowing what data it requires to do whatever it is it does, i.e. what other code it depends on, and knowing what effects it has on the system behavior. This may sound simple, but a function or method can easily grow to the point where it is difficult to see what the dependencies truly are, e.g. if a compound data type is provided as the input to a function, is every piece used? If so, how are those pieces created; what parts of the codebase are truly responsible for the input to the function? Knowing what code calls the function is not enough unless the call site is entirely responsible for the input.

While dependencies may be difficult to unravel, the opposite problem is often much worse. Knowing what effects a piece of code has on the system can be extremely difficult in commonly used languages such as Java and Python, as there is often little limiting what some function or method can do. A given method may perform a simple calculation on its input and return the results. It may also perform an HTTP request to some service and receive the results that way – with no indication that the system communicates with the outside world in this manner, nor that the system *depends* on said service. It could also modify global state, which potentially changes the behavior of all other parts of the codebase that interact with said state, despite there being no direct interaction between the different parts. This has been called "spooky action at a distance,"[7] a reference to what Einstein thought of quantum mechanics.

> Most large software systems, even if apparently well-engineered on a component-by-component basis, have proved to be incoherent as a

whole due to unanticipated long-range "weird interactions" among supposedly independent parts. [25]

Knowing what a piece of code does not do is, for the above reasons, difficult in many languages. In fact, while these two questions may appear very similar, it is often not the case that knowing the answer to one of them lets the programmer deduce the answer to the other.

Good code must then, somehow, communicate as much information to the programmer as possible. At the same time, throwing too much information at the programmer will not help. We have seen that legacy code is often if not always imperative code, and that the OOP paradigm has been an oft-tried solution for legacy code. Despite this, many current legacy code systems are written in OOP languages such as Java, and while books provide dozens of ways to write good OOP code, and ways to reduce the problems of legacy code, there does not seem to be any reason to believe that OOP is the ultimate solution to legacy code, nor that OOP code is inherently the best type of code.

Object-oriented is not the only way to write code. Functional programming (FP) is a programming paradigm that takes quite a different approach than OOP, and purely functional programming provides tools to assist the developer in writing what we have now defined to be "good" code. The next section provides an introduction to the area and its ideas.

4.1.4 Summary

Legacy systems are an ubiquitous problem, costing enormous amounts of money in maintenance and development costs. The definition can be summed up as a system which is difficult to understand, and some heuristics have been defined for identifying code that is or is not likely to be difficult to understand. "Good" code is taken to be code that is easy for a programmer to identify what it does with respect to the system behavior, and what it does not do; "bad," then, is the obvious antonym.

This definition of good code coincides with many of the characteristics of code written in a purely functional style, which the next section is concerned with.

4.2 Purely Functional Programming

This section introduces the purely functional programming (pure FP) paradigm, and its strengths. In short, pure FP enforces "referential transparency," which means that any piece of code can be replaced with the value it evaluates to, wherever it appears in the source code. This makes it easier to understand the code, any function can be reduced, on a cognitive level, to a black box that can be passed around and used without having to think about its contents. [10]

Pure FP achieves this by using immutable data structures, eliminating side effects in code, and using a powerful type system to express effectful computations (e.g. code that interacts with the user).

4.2.1 Functional Programming

The functional paradigm can be seen as a natural extension to the lambda calculus, a model of computation invented by Alonzo Church, where a small set of variable binding and substitution rules is used to express computations.

The Turing machine and lambda calculus models of computation are equivalent [24]. However, while the Turing machine model is largely an abstract concept that is useful for modeling computation, but less so in actually solving programming problems, there are several actively used languages that are, at their core, some type of lambda calculus. One example is the purely functional language Haskell. The Glasgow Haskell Compiler, the de facto standard Haskell implementation, compiles to a language based on System-Fω, a typed lambda calculus, as an intermediate language [15] [23].

The main tool in the lambda calculus is defining and applying functions; unsurprisingly, functions are the focus of FP. In this case, "function" is defined in the mathematical sense, as a mapping from inputs to outputs, rather than the sense of a function in e.g. the C programming language, where it is rather a series of commands for the computer to execute.

In FP, then, if any function f is given some input x and produces some output y, it must *always* be the case that $f(x) = y$. Thus, wherever $f(x)$ (for this given x) appears in the code, it can be replaced with y, without changing the program behavior whatsoever. Conversely, if we have that $g(a) = b$, but calling $g(a)$ prints a value to a console window, g is *not* a function in this sense, as replacing $g(a)$ with b in the code would change the program behavior by not printing to the console.

4.2.2 Purity

This characteristic, that a function always produces the same output given some input, including effects, is what defines a "pure" function. Pure functions are referentially transparent, and can thus be seen as black boxes. Conversely, an impure function, i.e. a function with side-effects, cannot be referentially transparent.

An "effect" does not have to be something like interacting with the user, or making an HTTP request. It also includes in-place mutation of data, querying the OS for a random number, throwing an exception — the list goes on. In a pure FP language such as Haskell, these effects are encoded in the language's type system, making it possible to write programs in fact actually *do* things, while enjoying the advantages that pure FP provides.

4.2.3 Advantages

The advantages of pure FP are largely concerned with having the compiler enforce program correctness. By encoding effects in types that are checked by the compiler, the programmer is prevented from writing code that has side-effects.

More generally, while writing a program in a pure functional language, the programmer is encouraged by the language and environment to write code that is reusable and easy to reason about [10].

Using a language with a powerful type system, it is also possible to write code in such a way that the semantics of the program is, to some extent, expressed in the types. This allows the compiler to enforce *semantic* program correctness. The programmer can construct transformations between data structures and compose them to produce a program, and said program is type-checked to ensure some degree of correctness.

Another boon bestowed by a powerful type system such as Haskell's, is that if a programmer can express their problem in the language of category theory, they gain access to 70 years of documentation concerning their problem. This may appear unlikely, but in fact category theory appears to provide tools to naturally express many problems encountered in software development. One example is constructing and combining 2D vector diagrams[26]; another, defining UIs in a declarative manner[9].

If the abstractions used can be expressed in the type system, the compiler can help prove that the program is correct.

4.3 Summary

Legacy code is largely a problem of understandable code, which in turn is closely related to the effects a piece of code performs when run.

This introduction of pure FP has shown that one of the main features is to eliminate side-effects of functions. A pure function must, by definition, tell the compiler — and the programmer — what it does. As important, a pure function cannot do anything else. Pure FP when combined with a powerful type system such as that of Haskell has many more benefits, providing the programmer with tools to express the semantics of the program in such a way that the compiler can ensure some level of correctness.

It does not seem a stretch to expect these features of pure FP to provide assistance when working with, or preventing, the problems of legacy code. The next chapter goes into detail how this hypothesis was tested.

Chapter 5

Method

The purpose of this thesis was to evaluate whether pure FP can be a tool for working with legacy codebases, and if pure FP tends to lead to code that is less likely to exhibit legacy problems.

This chapter begins with arguments in favor of the hypothesis, enumerating some features of pure FP that appear advantageous for limiting the potential problems associated with legacy code. Next, the programming project that is the core of the thesis, the Graph Genetics Browser (GGB) is introduced, followed by how the hypothesis was evaluated in the context of this project.

5.1 Functional Programming and Legacy Codebases

As was shown in the previous chapter, pure FP has many tools and features that are likely to limit the problems of legacy code. This is not the first time this has been considered, as can be seen in the following quote by Joe Armstrong, one of the creators of the Erlang language:

> I think the lack of reusability comes in object-oriented languages, not functional languages. Because the problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.
>
> If you have referentially transparent code, if you have pure functions — all the data comes in its input arguments and everything goes out and leave no state behind — it's incredibly reusable. — Joe Armstrong[20]

This section begins by introducing PureScript, the programming language used in the thesis project. It continues with providing examples of language features that are likely to reduce the legacy code-related problems and code patterns in general.

5.1.1 PureScript

PureScript (PS) is a purely functional programming language in the style of Haskell, that compiles to JavaScript (JS) [1]. PS is immutable by default, pure, and has an advanced static type system that gives the programmer many tools to increase productivity and program correctness. Each of these features will be examined further in the following sections.

5.1.2 Immutability

Some data being "immutable" means it does not change. PS being "immutable by default" means that all values that one work with when writing PS code are immutable — nothing can change[2]. Instead, if a function e.g. sorts a list, it creates a new sorted list, rather than modify the existing list. This ensures that all other parts of the program that uses the input list continue to function as they did before the sorting function was called. This would not have to be the case if the list changed in memory. The programmer never has to think about copying values; PS takes care of it.

Immutability by default reduces the reach of code by eliminating one prominent side-effect. It provides the programmer with absolute certainty that:

1. Inputs to a function are unchanged in that function
2. Calling a function on some value does not change that value

Mutation is one common side-effect in imperative programming, but far from the only one. The next section considers effectful programming in general.

5.1.3 Purity

A purely functional programming language does not only prevent the side-effect of mutating data in a function; it prohibits functions from causing *any* side-effect[3]. Some examples of effects follow.

A *partial* function is one that is not defined on all its inputs, e.g. a function that tries to access an out of bounds index on an array, whose type is given in block 1. If the given index is outside the array, the function explodes, for example with an exception. Hardly what the type says it will do, so ‘unsafeIndex‘ is not truly a function.

```
unsafeIndex :: forall a. Array a -> Int -> a
```

Listing 1: Type of unsafe array indexing function

Another effect is working with implicit state; another, retrieving data from an external source, or updating the user interface. All of these effects can be

[1] `http://www.purescript.org`

[2] There are some mutable data structures in PS, e.g. arrays that support in-place mutation for efficiency. These are separate from the immutable versions; transforming between the two must be done explicitly. As mutation is an effect, it too is captured by the type system and must be done in the ‘Eff‘ type; see # TODO add ref to Purity section!!

[3] In PureScript, functions can be made impure by improper use of the FFI, or by using "unsafe" functions such as ‘unsafeCoerce‘. This is not encouraged.

useful, possibly vital, for our programs, so it would not be desirable to discard them in the name of purity. Fortunately, PS provides tools to encode effects in the types of functions, allowing us to write pure yet effectful functions. The next section gives more details.

5.1.4 Static Types

PS has a powerful type system that makes it possible to describe much more of the semantics of the program in a way that the compiler can describe and check for us.

For example, the effect of partiality can be captured in the type system with the ‘Maybe‘ type, defined in code block 2. A value of type ‘Maybe Int‘ can contain either some ‘Int‘ wrapped in a ‘Just‘, or ‘Nothing‘. When writing functions that take a ‘Maybe‘ as input, the PS compiler will ensure that both possibilities are accounted for by failing with an error if something is missing.

```
data Maybe a
  = Just a
  | Nothing
```

Listing 2: The Maybe data type

With ‘Maybe‘ it is possible to write a safe, pure version of ‘index‘, see block 3. The effect of potential failure is captured in the function returning ‘Maybe a‘ rather than ‘a‘.

```
index :: forall a. Array a -> Int -> Maybe a
```

Listing 3: Type for safe array indexing function

In PS, another type is used to encapsulate so-called native effects, such as printing to the console, updating the UI, etc. This is the

```
class Functor f where
  map :: forall a b. (a -> b) -> f a -> f b
```

Listing 4: Functor typeclass definition

Often typeclasses are used to work with algebraic and category-theoretic abstractions, which provide a powerful way to write code that is both general and can be used to write code such that the compiler can check the semantics of our program in a way that is relevant to the actual program behavior.

Parametric polymorphism is reminiscent of generics in languages such as Java, but more powerful. Consider again the function in block 3. The type says that it works on ‘forall a. Array a‘; meaning it accepts any array, no matter what it contains. This is what it means for a function to be "parametrically polymorphic". Besides reducing code duplication by not having to write one indexing function for ‘Array Int‘, another for ‘Array Boolean‘ etc., parametric polymorphism also provides some additional knowledge about the function, by

enforcing that the function cannot do anything with some of its arguments other than pass them around. By looking at the type signature for 'index', we *know* that the output (if it is not 'Nothing') must come from the given array, as the function cannot create values of type 'a' from thin air.

A more extreme example is given in code block 5. This is the type of the identity function; the identity function is in fact the only function with this type. This is because it is parametrically polymorphic — the only thing 'id' can do with its argument is return it, there is literally nothing else that can be done.

```
id :: forall a. a -> a
```

Listing 5: The identity function

There is much more to PS' type system, but this covers most of what is used in the thesis project.

5.1.5 Summary

PureScript is an excellent example of a purely functional language, providing all of the features examined in the Theory chapter. As it compiles to JS and has good support for interoperating with JS, it is a natural candidate for investigating the viability of pure FP as applied to a legacy codebase written in JS. The next section presents the chosen legacy system, and how it will be extended.

5.2 Extending a Legacy System

In this section, Biodalliance, the legacy system that was extended, is described. The extent and nature of the changes, and more details of the resulting application, are also given.

5.2.1 Biodalliance

Biodalliance (BD) is an open source (BSD licensed) HTML5-based genome browser written in JavaScript (JS) [5]. It is fast, supports several data formats commonly used in bioinformatics, and the plots displayed can be configured and customized. Since it is HTML5-based, it can be embedded into any web page and does not require any special tools to be used. It also supports exporting images as SVG, which can be used as high quality figures in publications.

For GN2 we want a genome browser that supports these features. We also want to be able to add new features, however BD has shown itself to be difficult to work with and extend, for reasons earlier defined as legacy code problems[4]. BD has been in development since 2010, and as of December 2017 consists of 17.5k lines of JS code (sans comments and whitespace) split into 61 files[5]. The codebase has grown organically over time, and has become complex, with single

[4]The author previously worked on extending BD as part of Google Summer of Code 2016, and encountered some difficulties: `https://chfi.se/posts/2016-08-22-gsoc-final.html`

[5]Biodalliance's source code can be found on GitHub: `https://github.com/dasmoth/dalliance`

pieces of functionality (such as rendering data to the HTML5 canvas) being split into several large functions in different files, and less than clear flow of control.

We want many of the features of BD, such as its support for many file formats, which have required much time and effort to be implemented. However, we also want new features, but the time and effort required to implement them in BD is much greater than necessary due to the legacy aspect of BD's codebase.

The solution that was decided upon was to develop a new genome browser. This browser would allow embedding BD within it, and so be compatible with BD and support the desired features. However, it would be a separate application, and be independent of BD's baggage. This application is the Genetics Graph Browser, to which the next section is dedicated.

5.2.2 Genetics Graph Browser

The Genetics Graph Browser was written in PS. It begun as a tool for constructing BD-compatible data structures for creating new ways to plot data, but soon grew to a web application that embeds BD and Cy.js, and provides communication between the two.

Throughout its development, various legacy-type problems were encountered in BD, and avoided or solved in GGB. As GGB is written in PS, this was done using various "features" of FP, from the sections above. In this way, the GGB project can be seen as a case study in working with legacy code using purely functional programming.

The main features of GGB include:

- Working with JS APIs
- Configuration
- Units and types
- Rendering data to the screen
- Communication between BD and Cy.js
- Creating a purely functional UI containing a legacy JS app

Besides wrapping BD and providing genome browser functionality, one of the goals of GGB is to support exploratory data analysis of both genome-based data as well as graph-based, semantic web data. Where BD provides the genome browser, Cytoscape.js (Cy.js), a graph theory tool written in JS [8], is wrapped to provide the graph-network functionality.

GGB supports connecting BD and Cy.js data, letting the user configure interactions between the two browsers based on user interaction (e.g. updating the Cy.js graph upon clicking on a gene in BD).

The hypothesis of this thesis was then evaluated by examining if and how pure FP helped with these "legacy problems," both when fixing problems or improving how a problem was solved in BD, including the parts of GGB that interact with BD, as well as parts of GGB that are not related to BD, but solutions to which would likely end up exhibiting legacy-style problems.

PS was chosen as the language for the project for its support for interoperation with JS — PS can simply call JS functions and vice versa. However, PS also enforces a purely functional programming style, making it ideal for evaluating the hypothesis.

5.3 Summary

The hypothesis, that functional programming techniques can help when working with an existing legacy code base, as well as lead to code that is less likely to exhibit legacy code problems in the future, was tested by developing an application in PureScript that both interfaces with an existing legacy genome browser, and is intended to be a stand-alone browser in the future.

Identifying features whose implementation were already cause for concern in BD, or had the potential to be implemented in problematic ways in GGB, and if and how FP helped reduce or eliminate those problems, provides a lens through which it was possible to make some judgements as to the validity of the hypothesis.

Chapter 6

Results

In this chapter, the resulting product, that is, GGB, is presented. After a brief overview of the browser itself, the rest of the chapter consists of deeper dives into some of the parts of the code that is most relevant to the thesis objective.

6.1 The Completed Product

GGB, at the end of the thesis project, can be configured to display genome-based data tracks in an embedded BD browser, and show graph-based data in an embedded Cy.js browser. Configuration of these browsers is done by providing a single configuration object to GGB, which GGB verifies and uses to instantiate the embedded browsers. There is also basic support for configurable interactions between the browsers, making it possible for each browser to react to events in the other. Finally, the insides of GGB has modules for easily creating new ways to display data in BD, and works with genome-based data in a unit-conscious manner.

6.2 Interfaces Between PureScript and JavaScript

The Graph Genetics Browser embeds BD and Cy.js, which are both written in JS. To interact with their respective APIs, we must use PS's Foreign Function Interface (FFI).

6.2.1 Foreign Function Interface Introduction

PS's FFI works by creating a JS source file with the same name as the PS module which is going to be interfacing with the FFI, in which we define the JS functions which we will be calling from PS. This FFI module is imported into PS using the 'foreign import' keywords, and providing type signatures for the values we import.

The type signatures are not validated, and there are no guarantees that the FFI functions will work – the FFI is outside the type system. Code block 6 shows an example of an FFI function which takes two values and prints their sum. In PS, one would normally have to make sure it makes sense to add two

values before attempting to do so, likewise when transforming some value to a String, however JS has no such qualms.

```
exports.showStuff = function(a) {
    return function(b) {
        return function() {
            console.log(a + b);
        }
    }
}
```

Listing 6: Unsafe function that "adds" two values and prints the result to the browser console.

JS knows nothing about the types, however when defining an FFI function in PS, a type signature must be provided. Using the type in 7 limits using the ‘showStuff‘ function on strings, and returns an effect, making the function pure and behave reasonably.

```
foreign import showStuff
  :: forall eff.
     String
  -> String
  -> Eff (console :: CONSOLE | eff) Unit
```

Listing 7: A safe type signature the function defined in [[6]].

The following sections present the modules used to wrap the BD and Cy.js APIs.

6.2.2 Biodalliance

Using ‘foreign import‘ it is possible to define types corresponding to foreign data structures, as values for such a type can only be created with the FFI. To work with BD, a foreign type corresponding to instances of the BD browser is defined as in code block 8.

```
foreign import data Biodalliance :: Type
```

Listing 8: The data type corresponding to BD browser instances.

We also need an FFI function to wrap the BD browser constructor. This takes the browser constructor, another helper function, and the BD configuration as arguments:

```
foreign import initBDimpl
  :: forall eff.
     Fn3
     Foreign
     RenderWrapper
     BrowserConstructor
     (HTMLElement -> Eff (bd :: BD | eff) Biodalliance)
```

Listing 9: The FFI import for the BD browser constructor wrapper.

The output of the function is a continuation that takes an HTML element to place the BD browser in, and produces the effect to create and return the BD instance.

Biodalliance can produce events, and for GGB's event system we need to be able to attach a handlers to parse and transmit them. We create a newtype to wrap the events from BD (to make sure we don't use a raw event where it shouldn't be), and an FFI function that takes a BD instance and an effectful callback, returning an effect that attaches the callback.

```
newtype BDEvent = BDEvent Json

foreign import addFeatureListenerImpl
  :: forall eff a.
     EffFn2 (bd :: BD | eff)
     Biodalliance
     (BDEvent -> Eff eff a)
     Unit
```

```
exports.addFeatureListenerImpl = function(bd, callback) {
    bd.addFeatureListener(function(ev, feature, hit, tier) {
        callback(feature)();
    });
};
```

6.2.3 Cytoscape.js

Like BD, we define a foreign type for the Cy.js browser instance. We also have types for the Cy.js elements, collections, and a newtype wrapper for events. Note how the CyCollection type is a type constructor:

```
foreign import data Cytoscape :: Type

-- | Cytoscape elements (Edges and Nodes)
foreign import data Element :: Type

-- | A cytoscape collection of elements
foreign import data CyCollection :: Type -> Type
```

The Cy.js constructor is similar to BD's, except we don't need to pass any functions to it, as we have Cy.js as a dependency. We can provide a HTML element and an array of JSON objects to be used as the initial graph:

```
foreign import cytoscapeImpl
  :: forall eff.
     EffFn2 (cy :: CY | eff)
     (Nullable HTMLElement)
     (Nullable JArray)
     Cytoscape
```

'Nullable' is a type for dealing with 'null' in the FFI. We don't actually use 'cytoscapeImpl', instead we provide more idiomatic wrapper, so the user can use the more common and idiomatic 'Maybe':

```
cytoscape :: forall eff.
             Maybe HTMLElement
          -> Maybe JArray
          -> Eff (cy :: CY | eff) Cytoscape
cytoscape htmlEl els =
  runEffFn2 cytoscapeImpl
    (toNullable htmlEl) (toNullable els)
```

The Cytoscape.js instance can be worked with in multiple ways. Data can be added to the graph, retrieved from it, and deleted:

```
-- | Add a Collection of elements to the graph
graphAddCollection
  :: forall eff.
     Cytoscape
  -> CyCollection Element
  -> Eff (cy :: CY | eff) Unit

graphGetCollection
  :: forall eff.
     Cytoscape
  -> Eff (cy :: CY | eff) (CyCollection Element)

graphRemoveCollection
  :: forall eff.
     CyCollection Element
  -> Eff ( cy :: CY | eff) (CyCollection Element)
```

The graph layout can be controlled with the ‘runLayout‘ function, which takes a ‘Layout‘ value to update the Cy.js browser’s current layout:

```
-- | Apply a layout to the graph
runLayout :: forall eff.
             Cytoscape
          -> Layout
          -> Eff (cy :: CY | eff) Unit
```

‘Layout‘ is simply a newtype wrapper over ‘String‘. The native Cy.js layout function takes a ‘String‘ as an argument, and with this newtype wrapper we can both easily support all the layouts supported by Cy.js – easily adding more if appropriate – while staying type-safe.

```
newtype Layout = Layout String

circle :: Layout
circle = Layout "circle"
```

Events

Cy.js produces events in JSON format, a newtype wrapper is used to keep things safe (and improve readability of type signatures):

```
newtype CyEvent = CyEvent Json
```

The ‘onEvent‘ FFI function takes an event handler of type ‘CyEvent -> Eff a‘, and a ‘String‘ representing the type of event, e.g. "click" for adding a handler on click events. The function returns an effect that attaches the handler to the provided Cytoscape instance:

```
onEvent :: forall a.
           Cytoscape
        -> String
        -> (CyEvent -> Eff a)
        -> Eff Unit
```

```
exports.onEventImpl = function(cy, evs, callback) {
    cy.on(evs, function(e) {
        callback(e)();
    });
};
```

CyCollection

The ‘CyCollection‘ type is used to work with collections of elements in the Cytoscape.js browser. As it is implemented in Purescript as a ‘foreign data import‘, there is no way to create values of this type without using the FFI, e.g. with ‘graphGetCollection‘. Likewise all functions that manipulate ‘CyCollection‘ values must be implemented in terms of the FFI.

‘CyCollection‘ is a semigroup where the binary operation is taking the union of the two ‘CyCollections‘:

```
exports.union = function(a, b) {
    return a.union(b);
};
```

```
foreign import union
  :: forall e.
     Fn2
     (CyCollection e)
     (CyCollection e)
     (CyCollection e)

instance semigroupCyCollection :: Semigroup (CyCollection e) where
  append = runFn2 union
```

Another common interaction with a collection is extracting a subcollection. With ‘CyCollection‘, we can use the ‘filter‘ function for this:

```
-- | Filter a collection with a predicate
filter :: forall e.
          Predicate e
       -> CyCollection e
       -> CyCollection e
```

The FFI definition of ‘filter‘ uses the Cy.js API:

```
exports.filterImpl = function(pred, coll) {
    return coll.filter(pred);
};
```

The ‘Predicate‘ type is another newtype wrapper, this time of functions from the given type to Boolean. Since it’s a newtype, it can be provided to the FFI functions without unwrapping it.

```
newtype Predicate e = Predicate (e -> Boolean)
```

The Cytoscape.js API provides some basic predicates on elements, nodes, and edges. For example:

```
foreign import isNode :: Predicate Element
foreign import isEdge :: Predicate Element
```

‘Predicates‘ are ‘contravariant‘ in their argument, meaning they can be ‘contramapped‘ over, which can be seen as the opposite of normal, ‘covariant‘ functors. This is done by precomposing the ‘Predicate‘ with a function ‘(a -> e)‘. For example, if we have some ‘Predicate Json‘, i.e. a function from JSON values to Boolean, we can contramap the ‘elementJson‘ function over it, ending up with a ‘Predicate Element‘. This lets us filter the Cytoscape graph with all the powerful JSON parsing tools at our disposal.

```
hasName :: Predicate Json
hasName = Predicate f
  where f json = maybe false (const true) $ json ^? _Object <<< ix "name"

elemHasName :: Predicate Element
elemHasName = elementJson >$< hasName
```

‘Predicate‘ is also an instance of the ‘HeytingAlgebra‘ typeclass. This lets us combine ‘Predicates‘ using the normal Boolean logic combinators such as ‘&&‘ and ‘||‘:

```
namedNodeOrEdge :: Predicate Element
namedNodeOrEdge = (elemHasName && isNode) || isEdge
```

Tests ‘CyCollection‘ is unit tested to help ensure that the graph operations work as expected. For example, the edges and nodes from a graph should both be subsets of the graph:

```
let edges = filter isEdge eles
    nodes = filter isNode eles
when (not $ eles ‘contains‘ edges) (fail "Graph doesn’t contain its edges")
when (not $ eles ‘contains‘ nodes) (fail "Graph doesn’t contain its nodes")
```

Conversely, the union of the edges and nodes should be equal to the original graph, and this should be commutative:

```
(edges <> nodes) ‘shouldEqual‘ eles
(nodes <> edges) ‘shouldEqual‘ eles
(edges <> nodes) ‘shouldEqual‘ (nodes <> edges)
```

6.2.4 Summary

6.3 Safe Application Configuration

Software needs to be configurable. The Genetics Graph Browser (GGB) has many pieces that can and/or need to be configured by the user or system administrator. For example, what tracks are currently in the view.

There are also functions that need to be provided from the external JS, such as the Biodalliance browser constructor, and the wrapper for Purescript-defined renderers.

Configuration in standard JS solutions is not safe. A problem that can arise in JS is, if a configuration is given as a regular JS object (string, dictionary, etc.), and each configuration piece is simply assigned to the respective place in the application, there is risk of some subpiece being misconfigured, or simply missing. Worst case, the application can then crash.

6.3.1 Configuring Biodalliance

To give an idea of how configuration can take place in a legacy JS codebase, we look at BD. Many parts of BD can be configured, not just the tracks to display. All of this is provided by the user as a single JS object, and passed to the browser constructor which takes care of configuring the browser.

A very basic browser configuration could look like this:

```
var biodalliance = new Browser({
  prefix: '../',
  fullScreen: true

  chr:        '19',
  viewStart:  30000000,
  viewEnd:    40000000,

  sources:
    [{name: 'Genome',
      twoBitURI:  'http://www.biodalliance.org/datasets/GRCm38/mm10.2bit',
      desc: 'Mouse reference genome build GRCm38',
      tier_type: 'sequence'
    }]
});
```

This object contains configuration of basic browser functionality (the properties 'prefix', which is the relative URL for icons and such data, and 'fullScreen' which controls how the browser itself is rendered); initial browser state ('chr', 'viewStart', 'viewEnd', which together define the chromosome and range of base-pairs the browser displays at start); and an array of track source definitions ('sources'), which define what data to show, and how. In this case a mouse genome sequence is the only track.

There are many more options and ways to customize the browser itself, likewise there are many different kinds of sources, and ways to configure them. All configuration is provided as JS objects, and the configuration data is used in various functions to initialize objects, from the browser itself to the tracks it displays. Since these functions are the place where the configuration options are defined, it is easy to add new configuration options; for the same reason, there is no easy way to know what a configuration option does, nor what values are legal.

We'll have a brief look at some code smells in the configuration process, before moving on to the configuration of GGB, and how it is implemented.

Code Examples

Defaults and browser state BD has many features, and makes use of a lot state in its main Browser object to function. Due to this, much of the browser construction sets various pieces of state to default values, and sets others to values from the configuration, or uses some configuration data to create an initial value.

As a prime example of primitive obsession, nearly all of the fields set by the browser constructor are numbers or strings, with only a few objects. Since this is JavaScript, there is no real type checking, though some of the code makes use of basic validation. However, it is wordy, and does not provide much:

```
if (opts.viewStart !== undefined &&
    typeof(opts.viewStart) !== 'number') {
    throw Error('viewStart must be an integer');
}

this.viewStart = opts.viewStart;
```

Verbosity alone makes it is understandable that only a few of the many values set are checked like this. After various defaults are set, all other options from the configuration object are set on the browser object:

```
// 140 lines of setting default options
for (var k in opts) {
    this[k] = opts[k];
}
```

Meaning if the user has somehow set an option with the same name as one of the fields used by BD, the browser will silently use the provided value, even though it may be entirely incorrect. E.g. providing a number to a function expecting a HTML DOM element.

Entangled types (probably remove this) Sources and styles are both tightly coupled and separate – problematic...

Neat solutions in PS include mapping 'Source' -> 'SourceSansStyle', Modelling type as (Source, Style), etc. (Isomorphic)

```
var sourcesAreEqual =
      sourcecompare.sourcesAreEqual;
var sourcesAreEqualModuloStyle =
      sourcecompare.sourcesAreEqualModuloStyle;
```

Indirection The constructor itself calls a method 'browser.realInit()' as it finishes. This function continues much like the constructor, preparing the browser. Finally, the method 'browser.realInit2()' is called.

Basically, the entire constructor, and its "subroutines" realInit and realInit2 (those names are themselves code smells), create ad-hoc browser elements, set a bunch of default state, some of which are just values, others are derived from other configuration or data, until the whole thing is "ready".

Event handlers The browser constructor also sets the various DOM event handlers used by the browser's UI. A handler can be any function that takes an event as argument, meaning it is easy to write a handler that directly takes care of e.g. updating a UI element. That is also a problem, as every handler that modifies some UI state is another possibility for interference when working with the UI.

#+begin$_{\text{comment}}$

Validation and transformations This style of code is commonly seen in code throughout BD, including configuration:

```
while (sti < st.length &&
       ry > st[sti].height &&
       sti < (st.length - 1)) {
    ry = ry - st[sti].height - tier.padding;
    ++sti;
}
if (sti >= st.length) {
    return;
}
```

This code removes the sum of the height in pixels of the tracks from a value. It does this with external effects and state.

#+end$_{\text{comment}}$

Code smell summary There's no control that each part of the configuration/construction works as it should, nor is there any structure to it. These functions:

- Create and work with HTML elements
- set default options, configuration
- setting a whole lot of UI state, including that which is used in submenus etc.
- Sets event handlers, which are filled with code duplication, low level handling of events, low level responses to events. Scrolling up and down with the keys is a good example: The **same** code, 80 lines long, duplicated, right after another.

Another approach

The solution used in GGB is to parse the configuration at the start of the program, from a raw Javascript JSON object into a Purescript type, with validation and error handling and reporting. For this I opted for purescript-foreign and purescript-argonaut, annotating all failures with error messages, which bubble up to the main configuration parser, which returns an error object or a successfully parsed configuration.

6.3.2 BrowserConfig

The type BrowserConfig represents the highest level of the GGB configuration hierarchy; it is the parsed version of the JS object provided by the user. This is the definition:

```
newtype BrowserConfig =
  BrowserConfig
    { wrapRenderer :: RenderWrapper
      , bdRenderers :: StrMap RendererInfo
      , browser :: BrowserConstructor
      , tracks :: TracksMap
      , events :: Maybe
          { bdEventSources :: Array SourceConfig
          , cyEventSources :: Array SourceConfig
          }
    }
```

At this point, the specific types of the values in the record are irrelevant; the important part is that they're all Purescript types, and have been parsed and validated. The parsing is done by the parseBrowserConfig function, which has the following type signature:

```
parseBrowserConfig :: Foreign -> F BrowserConfig
```

NOTE: add link to source, ideally make parseBrowserConfig and BrowserConfig clickable, or add links below the script (you could generate them from Emacs tags). Also make sure this code passes the current version. Same for all others. Note that this will be your documentation too.

parseBrowserConfig is a function that reads a JS object containing the necessary information to start the GGB, for example which tracks are included in the view, and functions for interfacing with BD.

The pattern 'Foreign -> F a' really says that a function named parseBrowserConfig is applied to Foreign type F and returns a BrowserConfig. This type of action is ubiquitous in the modules concerning configuration, because we use the library 'purescript-foreign'. The type 'Foreign' is part of Purescript and is simply anything that comes from outside Purescript, and thus must be parsed before any information can be extracted from them. 'F' is a type synonym:

```
type F = Except (NonEmptyList ForeignError)

data ForeignError =
    JSONError String
  | ErrorAtProperty String ForeignError
  | ErrorAtIndex Int ForeignError
  | TypeMismatch String String
  | ForeignError String
```

'Except' is practically 'Either', and lets us represent and handle exceptions within the type system. In this case, the error type is a non-empty list of these possible error values. If something has gone wrong, there is at least one error

message connected to it; it is simply impossible to fail a parse without providing an error message!

From the type signature, then, we see that the function name does not lie: it does attempt to parse Foreign data into BrowserConfigs, and must fail with an error otherwise. We know this, because the function does not have access to anything other than the raw configuration data, which means all the pieces of the completed BrowserConfig must be extracted from the provided configuration, or there are default values provided in the function itself.

Let's look at one of the lines from the function definition (note: if you are new to Purescript the syntax may look strange - ignore the details, it will slowly make sense and you may appreciate the terseness in time).

```
parseBrowserConfig :: Foreign -> F BrowserConfig
parseBrowserConfig f = do
  browser <- f ! "browser" >>= readTaggedWithError "Function" "Error on 'browser':"
```

'F' is a monad, which in this case is simply an object containing state (Either a NonEmptyList or an error), so what is happening here is first an attempt to index into the "browser" property of the supplied Foreign value, followed by an attempt to read the Javascript "tag" of the value. If the tag says the value is a function, we're happy and cast the value to the type BrowserConstructor bound to the name browser, which is later referred to when putting the eventual BrowserConfig together. If the object doesn't have a "browser" property, or said property is not a JS function, we fail, and tell the user what went wrong.

6.3.3 Tracks

Tracks configurations are different for BD tracks and Cy.js graphs, though both are provided as arrays of JSON, under different properties in the 'tracks' property of the configuration object, they are treated in their respective sections.

Biodalliance

Tracks using BD are configured using BD source configurations; they are directly compatible with Biodalliance configurations. Because of this, there is little validation on these track configurations, as there would be no reasonable way of representing the options in Purescript, as they are spread out over the entire BD codebase. There are, for example, numerous properties which can describe from where the track will fetch data and what kind of data it is, which are logically disjoint but nevertheless technically allowed by Biodalliance (though likely with undesired results).

So, the GGB takes a hands-off approach to BD tracks, and the only validation that takes place is that a track must have a name. If it does, the JSON object is later sent, unaltered, to the Biodalliance constructor.

The Biodalliance constructor is another parameter that the configuration requires. This and the 'wrapRenderer :: RenderWrapper' function are required for the BD interface to function properly, and are JS functions provided by Biodalliance.

Cytoscape.js

Cytoscape graphs are currently configured by providing a name and a URL from which to fetch the elements in JSON format.

#+begin$_{\text{comment}}$

6.3.4 Events

When a user interacts with a track, e.g. by clicking on a data point, the track can communicate the interaction to the rest of the system, including other tracks. The user can configure the structure of the events that a track produces, and what a track does when receiving an event of some specific structure, e.g. scrolling the track on receiving an event containing a position.

TODO: remove below text into the source files for documentation. You can refer to that, but I would just continue with TrackSink here.

Parsing the user-provided SourceConfigs The SourceConfig and TrackSource validation is done in Either String, while the BrowserConfig parsing is done in the type Except (NonEmptyList ForeignError). To actually use these functions when parsing the user-provided configuration, we need to do a transformation like this:

```
toF :: Either String ~> Except (NonEmptyList ForeignError)
```

Fortunately, Either and Except are isomorphic - the difference between the two is only in how they handle errors, not what data they contain. There already exists a function that does part of what we need:

```
except :: forall e m a. Applicative m => Either e a -> Except e a
```

Now we need a function that brings Either String to Either (NonEmptyList ForeignError). We can use the fact that Either is a bifunctor, meaning it has lmap:

```
lmap :: forall f a b c.
        Bifunctor f
     => (a -> b)
     -> f a c -> f b c
```

It's exactly the same as map on a normal functor, except it's on the left-hand type.

The bifunctor instance on Either can be seen as letting us build up a chain of actions to perform on both success and failure, a functional alternative to nested if-else statements.

The final piece we need is a way to transforming a String to a (NonEmptyList ForeignError). Looking at the definition of the ForeignError type, there are several data constructors we could use. Easiest is (ForeignError String), as it simply wraps a String and doesn't require any more information. To create the NonEmptyList, we exploit the fact that there is an Applicative instance, and use ‘pure‘:

```
f :: String -> NonEmptyList ForeignError
f = pure <<< ForeignError
```

Putting it all together, we have this natural transformation:

```
eitherToF :: Either String ~> F
eitherToF = except <<< lmap (pure <<< ForeignError)
```

Now we can parse the events configuration in the BrowserConfig parser:

```
events <- do
  evs <- f ! "eventSources"

  bd <- evs ! "bd" >>= readArray >>= traverse parseSourceConfig
  cy <- evs ! "cy" >>= readArray >>= traverse parseSourceConfig

  _ <- eitherToF $ traverse validateSourceConfig bd
  _ <- eitherToF $ traverse validateSourceConfig cy

  pure $ Just $ { bdEventSources: bd
                , cyEventSources: cy
                }
```

Note how we discard (_ <- ...) the results from the config validation; we only care about the validation error, since the configuration values have already been parsed.

Future work Typing events – types are there, just not checked (also only makes sense w/ some kinda DSL/interpreter)

#+end$_{\text{comment}}$

6.4 Working With Units

6.4.1 Biodalliance/The problem

It is often the case that values in programs are represented using primitive types, rather than using the fact that different units in fact can be viewed as different types. When all values are regular JS numbers, there is nothing to stop the programmer from accidentally adding a length to a weight, which is likely to lead to problems. It also becomes more difficult to comprehend what a piece of code does.

While they are displayed in visualizations, graphs, etc., the underlying representation is rarely anything other than a string or a number. That is, to the computer, there is no semantic difference between e.g. the position of a basepair on some chromosome, the volume of a house, or pi – all of these numbers could be used interchangeably.

BD uses mainly raw JS numbers and strings for representing its state and data, with a few JS objects used mainly for more complex information.

One way to solve this problem in JS would be to use something like the ‘daggy‘ library, which adds tagged sum "types" to JS. The developer still needs to make sure they are used correctly, but at least the program will fail with an error if a value representing a pixel length is supplied to a function expecting a length in basepairs.

Since Purescript actually has a type checker and built-in sum types, we expect it to be easier to represent these kinds of units. In fact, Purescript lets us create new types wrapping existing types, without any runtime cost. This is done using newtypes.

6.4.2 Newtypes

Newtypes are one of the ways of creating types in Purescript. They can only have one single data constructor, and that constructor must have one single parameter, hence the intuition that they wrap an existing type. At runtime, values in a newtype are identical to values of the underlying type, which can be exploited when working with the FFI, plus there is no performance hit when using newtypes.

6.4.3 Positions

Biodalliance uses basepairs (Bp) for all position data, and stores this data as a regular Javascript Number value. It's not uncommon for data to provide its position information in megabasepairs (MBp). Obviously, treating a Bp as an MBp, or vice versa, leads to problems, but if it's just a Number being thrown around, there's no way to avoid the problem other than trusting the programmer and user to do things correctly.

As a programmer and user, I find the idea of doing so reprehensible, hence the Bp and MBp newtypes:

```
newtype Bp = Bp Number
newtype MBp = MBp Number
```

To work with these, we can use pattern matching:

```
toBp :: Number -> Bp
toBp x = Bp x

fromBp :: Bp -> Number
fromBp (Bp x) = x
```

However, Purescript provides a typeclass to minimize this boilerplate, namely the 'Newtype' typeclass. The compiler derives the instance, and we can then use the generic 'wrap' and 'unwrap' functions:

```
derive instance newtypeBp :: Newtype Bp _
derive instance newtypeMBp :: Newtype MBp _

mbpToBp :: MBp -> Bp
mbpToBp x = wrap $ (unwrap x) * 1000000.0
```

Purescript also provides facilities for deriving typeclass instances for newtypes. Deriving the typeclasses used in arithmetic lets us use normal operators when working with Bp and MBp:

```
derive newtype instance eqBp :: Eq Bp
derive newtype instance ordBp :: Ord Bp
derive newtype instance fieldBp :: Field Bp
derive newtype instance euclideanRingBp :: EuclideanRing Bp
derive newtype instance commutativeRingBp :: CommutativeRing Bp
derive newtype instance semiringBp :: Semiring Bp
derive newtype instance ringBp :: Ring Bp

-- now we can do
p1 = Bp 123.0
p2 = Bp 400.0

-- p1 + p2 == Bp 523.0
```

6.4.4 Chromosomes

Biodalliance represents chromosome identifiers as strings. Like with Bp, a newtype wrapper helps keep track of things:

```
newtype Chr = Chr String
derive instance newtypeChr :: Newtype Chr _
derive newtype instance eqChr :: Eq Chr
derive newtype instance ordChr :: Ord Chr
derive newtype instance showChr :: Show Chr
```

6.4.5 Scale

When drawing data to the screen, we need to be able to transform between screen coordinates and the coordinates used by data. For simplicity's sake, we only care about mapping between basepairs and pixels. We represent this with another newtype wrapping Number:

```
newtype BpPerPixel = BpPerPixel Number
derive instance newtypeBpPerPixel
  :: Newtype BpPerPixel _

bpToPixels :: BpPerPixel -> Bp -> Number
bpToPixels (BpPerPixel s) (Bp p) = p / s

pixelsToBp :: BpPerPixel -> Number -> Bp
pixelsToBp (BpPerPixel s) p = Bp $ p * s
```

6.4.6 Features

In BD, a ‘feature‘ is basically any data point. While the feature objects in BD can become arbitrarily complex as various data parsers construct them in different ways, there are only three minimal pieces of information required: what chromosome the feature is on, and what range of basepairs on the chromosome it covers.

In Purescript, we represent this type as an algebraic data type (ADT).

```
data Feature c r = Feature Chr c c r
```

For convenience, we let the compiler derive how to compare two ‘Features‘ for equality and order:

```
derive instance eqFeature :: (Eq c, Eq r) => Eq (Feature c r)
derive instance ordFeature :: (Ord c, Ord r) => Ord (Feature c r)
```

There is also a smart constructor for creating ‘Features‘ only with coordinates that can be transformed to basepairs.

```
feature :: forall c r.
           HCoordinate c
        => Chr -> c -> c -> r
        -> Feature c r
feature = Feature
```

Since ‘Feature‘ has two type parameters, one for the coordinates and one for the data, and is covariant in both, we have a bifunctor instance:

```
instance bifunctorFeature
  :: Bifunctor Feature where
    bimap f g (Feature chr xl xr r) =
      Feature chr (f xl) (f xr) (g r)
```

6.5 Transforming Data for Screen and Space

A "glyph" is something that can be drawn to the browser display, as well as be exported to SVG. They are also what the user interacts with, and so have bounding boxes that are used to detect whether the user has clicked on them, to produce browser events.

6.5.1 Biodalliance

Biodalliance has a number of Glyphs, which are classes sharing a basic interface – they have a function which draws itself to the canvas, one which produces an SVG element, as well as functions providing the bounding boxes.

The Glyphs range from basic geometric shapes such as boxes and triangles, to more complex "higher order" ones, which take other glyphs and e.g. translates them.

Glyphs are created using the appropriate constructor, which takes data such as position on screen, color, and so on. For example, the Glyph to create a box (rectangle) requires position, size, color, transparency, and radius (for rounding corners):

```
function BoxGlyph(x, y,
                  width, height,
                  fill, stroke,
                  alpha, radius) {
    this.x = x;
    this.y = y;
    this._width = width;
    this._height = height;
    this.fill = fill;
    this.stroke = stroke;
    this._alpha = alpha;
    this._radius = radius || 0;
}
```

These fields are then used in the other methods, such as ‘draw()‘, which draws the Glyph to a provided canvas context. All of the ‘draw()‘ methods use basic HTML5 canvas commands. A snippet of ‘BoxGlyph.draw()‘ follows; the function argument is a HTML5 canvas context to perform the drawing actions on:

```
BoxGlyph.prototype.draw = function(g) {
    var r = this._radius;
  // ...
    if (this._alpha != null) {
        g.save();
        g.globalAlpha = this._alpha;
    }

    if (this.fill) {
        g.fillStyle = this.fill;
        g.fillRect(this.x, this.y,
                   this._width,
                   this._height);
    }

    if (this.stroke) {
        g.strokeStyle = this.stroke;
        g.lineWidth = 0.5;
        g.strokeRect(this.x, this.y,
                     this._width,
                     this._height)
    }

    if (this._alpha != null) {
        g.restore();
    }
}
```

Note that the HTML5 canvas context is stateful, and commands such as "fillRect" and "strokeRect" draw shapes using the current state, which is set with e.g. the ‘fillStyle‘ and ‘strokeStyle‘ fields.

BD supports exporting the browser view to SVG, which is accomplished by each Glyph having a ‘toSVG‘ method. ‘toSVG()‘ returns an SVG element representing the glyph in question.

```
BoxGlyph.prototype.toSVG = function() {
    var s = makeElementNS(NS_SVG, 'rect', null,
                          {x: this.x,
                           y: this.y,
                           width: this._width,
                           height: this._height,
                           stroke: this.stroke || 'none',
                           strokeWidth: 0.5,
                           fill: this.fill || 'none'});

    if (this._alpha != null) {
        s.setAttribute('opacity', this._alpha);
    }

    return s;
}
```

Constant functions ‘min()‘, ‘max()‘, and so on, are used to calculate the bounding boxes of glyphs, to detect whether a user has clicked on a glyph:

```
BoxGlyph.prototype.min = function() {
    return this.x;
}

BoxGlyph.prototype.max = function() {
    return this.x + this._width;
}

BoxGlyph.prototype.height = function() {
    return this.y + this._height;
}
```

The problems with this way of creating and working with glyphs largely relate to code duplication, and it being difficult to compose existing glyphs to create new ones. Having to explicitly write these various functions provide many opportunities for mistakes to sneak their way in.

In GGB, we instead use a Free monad to provide a simple DSL for describing Glyphs. The DSL is then interpreted into functions for rendering it to a HTML5 canvas, as an SVG element, and bounding boxes. This vastly reduces the places where mistakes can be made, and also makes it easy to test – the canvas rendering code need only be written once and then used by all Glyphs, and it can be tested on its own.

6.5.2 Glyphs in the Genetics Graph Browser

We require some types to represent our Glyphs. First, a simple ‘Point‘ type representing a point in 2D space, and the ‘GlyphF‘ type which contains the

commands in our Glyph DSL:

```
type Point = { x :: Number, y :: Number }

data GlyphF a =
    Circle Point Number a
  | Line Point Point a
  | Rect Point Point a
  | Stroke String a
  | Fill String a
  | Path (Array Point) a
```

The type parameter ‘a‘ in ‘GlyphF‘ is there so we can create a Functor instance. This is important, because the Free monad wraps a Functor. To reduce boilerplate, we let the compiler derive the Functor instance for GlyphF – if a type can be made into a Functor, there is only one implementation, and it is mechanical.

```
derive instance functorGlyph :: Functor GlyphF
```

The Free monad is named so because it is the
In Haskell, the definition is very simple, thanks to non-strict evaluation:

```
data Free f a = Pure a
              | Bind f (Free f a)
```

Here, ‘f‘ is the underlying functor, and ‘a‘ is whatever value we want to return. ‘Free‘ provides two value constructors; one containing only a single value (equivalent to the ‘pure‘ function in the Applicative typeclass), the other containing a value in our underlying functor, which in turn contains the next "step" in the computation in the Free monad. The Free monad can be seen as a list of commands in a DSL, where said DSL is defined entirely in the underlying functor. Another way of looking at it is as a list of functors. In fact, if the underlying functor is ‘(,) a‘, that is, the type of two-element pairs where the first element is of some type ‘a‘, we have a type that is isomorphic to a regular list:

```
type List a = Free ((,) a) ()

[1,2,3] ~ Bind (1,
             Bind (2,
                Bind (3, (Pure ()))))
```

The Purescript definition of Free is more complicated, so as to be stack-safe in a strict language. However, the rest of the code is in Purescript.

The free monad constructs a list of commands, and these commands can then be interpreted into some other functor, including effectful ones. Examples will come; there is some work left before we get there. First we wrap our ‘GlyphF‘ functor in ‘Free‘, with a type synonym to make things cleaner:

```
type Glyph = Free GlyphF
```

Next we want to lift our ‘GlyphF‘ data constructors into functions. This is done using the ‘liftF‘ function, which has the following signature:

```
liftF :: forall f a. f a ~> Free f a
```

Here we use ‘liftF‘ to lift two of the commands in ‘GlyphF‘ to ‘Free GlyphF‘, the rest are exactly analogous and elided:

```
circle :: Point -> Number -> Glyph Unit
circle p r = liftF $ Circle p r unit

line :: Point -> Point -> Glyph Unit
line p1 p2 = liftF $ Line p1 p2 unit

stroke :: String -> Glyph Unit
stroke c = liftF $ Stroke c unit

-- ..
```

Since it’s a monad, we also can use do-notation to create glyphs, after creating some helper functions:

Now we have a number of functions which produce values in the type ‘Free GlyphF‘. With them, we can use Purescript’s do-notation, and all the other tools that come with the Monad typeclass. As an example, in code block 6.5.2 we create a simple glyph consisting of a red ‘X‘ over a black circle.

```
crossedOut :: Point -> Number -> Glyph Unit
crossedOut p@{x,y} r = do
  stroke "black"
  circle p r
  stroke "red"
  line {x:x-r, y:y-r} {x:x+r, y:y+r}
  line {x:x-r, y:y+r} {x:x+r, y:y-r}
```

A Glyph, then, is simply a data structure. The interesting part lies in interpreting this data structure; or, in other words, transforming it into another data structure, especially one that performs effects. In fact, an interpreter

consists of a natural transformation from the ‘GlyphF‘ functor to some other functor.

We continue with a simple interpreter, one which transforms a ‘Glyph‘ into a ‘String‘, which can then be printed to console, or otherwise logged.

6.5.3 Logging glyphs

The GlyphF.Log interpreter transforms Glyphs to Strings, which we can then log to the console. To run an interpreter, we use ‘foldFree‘; see 6.5.3 for its type signature.

```
foldFree :: forall f m.
            MonadRec m
         => (f ~> m)
         -> (Free f)
         ~> m
```

The ‘MonadRec‘ constraint ensures that only monads supporting tail recursion can be used. Without it stack safety would be a problem. The type operator ~> denotes a natural transformation, it has the same meaning as:

```
forall a. f a -> g a
```

That is, it is parametrically polymorphic mapping between functors, and so cannot touch the contents of the functor.

For producing a String, the Writer type is a natural fit, and conveniently also has a MonadRec instance. The type of the natural transformation is then:

```
glyphLogN :: GlyphF ~> Writer String
```

The definition of the function is also simple enough. For each primitive, write an appropriate string, and return the contents of the functor:

```
glyphLogN (Stroke c a)   = do
  tell $ "Set stroke style to " <> c
  pure a

glyphLogN (Circle p r a) = do
  tell $ "Drawing circle at ("
      <> show p.x <> ", "
      <> show p.y
      <> ") with radius "
      <> show r <> "."
  pure a
```

Running the interpreter consists of applying this natural transformation to the Free GlyphF, using foldFree, and then getting the resulting String from the Writer. The function ‘showGlyph‘ nearly writes itself at this point, as seen in code block 6.5.3.

```
execWriter :: forall w a. Writer w a -> w
```

```
showGlyph :: forall a. Glyph a -> String
showGlyph = execWriter <<< foldFree glyphLogN
```

For example, logging the process of drawing the previously defined ‘crossedOut‘ glyph at the point ‘{ x: 40.0, y: 10.0 }‘ with radius ‘3.0‘ would produce the output seen in block 6.5.3.

```
Drawing circle at (40.0, 10.0) with radius 3.0
Drawing line from (37.0, 7.0) to (43.0, 13.0)
Drawing line from (37.0, 13.0) to (43.0, 7.0)
```

6.5.4 Drawing glyphs to canvas and SVG

When drawing to canvas, we use Eff as the target for our natural transformation. Interpretation is done by performing the appropriate canvas effects, see code block 6.5.4. ‘glyphEffN‘ is then used in ‘renderGlyph‘, in block 6.5.4, to interpret an entire ‘Glyph‘ structure into a canvas instruction.

Code can be found on Github.

```
import Graphics.Canvas as C
import Graphics.Canvas (CANVAS, Context2D)
```

```
glyphEffN :: forall eff.
             Context2D
          -> GlyphF
          ~> Eff (canvas :: CANVAS | eff)
glyphEffN ctx (Stroke c a) = do
  _ <- C.setStrokeStyle c ctx
  pure a
glyphEffN ctx (Circle p r a) = do
  _ <- C.beginPath ctx
  _ <- C.arc ctx { x: p.x
                 , y: p.y
                 , r: r
                 , start: 0.0
                 , end: 2.0 * Math.pi
                 }
  _ <- C.stroke ctx
  _ <- C.fill ctx
  pure a
-- ..
```

```
renderGlyph :: forall eff.
               Context2D
            -> Glyph
            ~> Eff (canvas :: CANVAS | eff)
renderGlyph = foldFree <<< glyphEffN
```

SVG on the other hand interprets ‘Glyphs‘ into the ‘SVG‘ type, a monad transformer stack defined in block 6.5.4.

```
import Graphics.SVG (SVG, SVGElement)
import Graphics.SVG as SVG
```

```
type SVG a =
  StateT SVGContext
    (Writer (Array SVGElement)) a
```

The result is a series of commands which can be used to produce the desired SVG element. The interpreter is in block 6.5.4, and is very similar to the HTML canvas interpreter in block 6.5.4.

```
interpSVGEff :: GlyphF ~> SVG
interpSVGEff (Stroke c a)  = do
  SVG.setStrokeStyle c
  pure a
interpSVGEff (Circle p r a) = do
  SVG.circle p.x p.y r
  pure a
-- ..
```

The interpreter is used in code block `code:glyph-draw-5`, first to map ‘Glyphs‘ to pure SVG elements, then to render the SVG elements using the DOM.

6.5.5 Generating bounding boxes

BD produces events when clicking on glyphs, events that GGB makes use of. To do this, BD expects four constant functions on each glyph. In Purescript, the "bounding box" type would look like ‘BoundingBox‘ in code block 6.5.5. Since ‘BoundingBox‘ is a record, it has the exact same runtime representation that BD expects.

```
type BoundingBox =
  { min :: Unit -> Number
  , max :: Unit -> Number
  , minY :: Unit -> Number
  , maxY :: Unit -> Number }
```

When constructing glyphs in BD, each new glyph provides its own explicit bounding box. This is clearly insufficient for our purposes; instead, we make use of the fact that bounding boxes form a semigroup, and in fact also a monoid. A brief introduction of these concepts follows.

Semigroups and monoids

Semigroups and monoids are concepts from abstract algebra and category theory, however they are immensely useful in pure FP, as they appear in many different areas.

A semigroup is an algebraic structure consisting of a set together with an associative binary operation. Let ‘S‘ be the set in question and ‘x‘, ‘y‘, ‘z‘ any three elements from ‘S‘, with the binary operation ‘<>‘. If this following law is true, we have a semigroup:

Associativity : $(x <> y) <> z == x <> (y <> z)$

Semigroups can intuitively be viewed as things that can be "appended" to each other. For example, arrays, lists, and strings are semigroups, with the binary operation being appending the two arguments. Another example is the natural numbers with addition as the operation.

A monoid is a semigroup with one special element, an identity. The example from above is a monoid if there is an element ‘e‘ in ‘S‘ such that these laws apply for all elements ‘x‘ in ‘S‘:

$Leftidentity : x <> e == x$ $Rightidentity : e <> x == x$

Again, arrays, lists, and strings are monoids, where the identity element is the empty array/list/string. The natural numbers with addition form a monoid only if zero is counted among the naturals; without zero, it is only a semigroup.

With these definitions we can explore how bounding boxes form a monoid.

Monoidal bounding boxes

The type corresponding to a glyph's position is ‘GlyphPosition‘ in block `code:glyph-bounding-box-2` It is a newtype wrapper over a record describing each of the four edges of the bounding box. This is a semigroup, where the binary operation produces the minimal bounding box that covers both inputs. That is, we take the minimum or maximum of the respective values, to get whichever maximizes the area covered:

Note the use of the the minimum and maximum functions from the Math module, and how they're really doing all the heavy lifting. For ‘GlyphPosition‘ to be a monoid, we require an identity element. We can use the fact that the semigroup instance uses ‘min‘ and ‘max‘ as a hint. While there is no minimum or maximum real number, we can cheat and use positive and negative infinity, which exist in JS. Then we have:

```
forall x. Math.min x  infinity == x
forall x. Math.max x -infinity == x
```

Now the identity ‘GlyphPosition‘ is obvious – the minimum sides are set to positive infinity, and the maximum sides are set to negative infinity:

Now, with our ‘Monoid‘ instance in hand, we can write another interpreter for Glyph, using Writer as our monad in the natural transformation, see code block `code:glyph-bounding-box-6`.

Finally, in block `code:glyph-bounding-box-7` this interpreter is used exactly as the previous Writer-based interpreters were.

Now bounding boxes come for free with all ‘Glyphs‘.

Testing our monoid

Semigroups and monoids have laws; while I'm reasonably confident in having created a Real Monoid, I prefer to have my computer make sure. To do this, I use purescript-jack, a property-based testing framework, like QuickCheck.

First, some utility functions to generate and render GlyphPositions:

```
type ThreeGlyphs =
  { l :: GlyphPosition
  , c :: GlyphPosition
  , r :: GlyphPosition}

renderGlyphs :: ThreeGlyphs
             -> String
renderGlyphs {l,c,r} =
     "{ l: " <> show l
  <> ", c:"  <> show c
  <> ", r:"  <> show r <> "}"

genGlyphPosition :: Gen GlyphPosition
genGlyphPosition = do
  let cf = toNumber <$>
             chooseInt (-10000000) (10000000)
  min <- cf
  max <- cf
  minY <- cf
  maxY <- cf
  pure $ GlyphPosition { min, max, minY, maxY }

genThreeGlyphs :: Gen ThreeGlyphs
genThreeGlyphs = do
  l <- genGlyphPosition
  c <- genGlyphPosition
  r <- genGlyphPosition
  pure $ {l, c, r}
```

The law all semigroups should abide is associativity. In Jack, we describe a Property asserting that parentheses don't matter for equality:

```
prop_semigroup :: Property
prop_semigroup =
  forAllRender renderGlyphs genThreeGlyphs \pos ->
    property $ (pos.l <> pos.c) <> pos.r == pos.l <> (pos.c <> pos.r)
```

In addition to that, monoids require that the identity element in fact be left and right identity. The Property:

```
prop_monoid :: Property
prop_monoid =
  forAll genGlyphPosition \pos ->
    property $ (pos <> mempty == pos) &&
               (mempty <> pos == pos)
```

Jack then takes care of generating GlyphPositions, ensuring that these properties hold.

6.5.6 Putting it all together

With these interpreters, we can create a function that produces a JS object that is compatible with BD. BD expects a glyph to have:

- a function to draw the glyph to a provided canvas
- a function to export the glyph to SVG
- functions that provide the bounding box
- optionally the relevant feature, or data point, that was used to produce the glyph

To do this, we exploit the fact that Purescript records are JS objects, by constructing a record with the appropriate properties, and transform it to a Foreign value. The main function in its entirety:

```
writeGlyph' :: forall a c r.
               Maybe (Feature c r)
            -> Glyph a
            -> Foreign
writeGlyph' f g =
  toForeign
    { "draw": unsafePerformEff <<< \ctx -> Canvas.renderGlyph ctx g
    , "min": const p.min
    , "max": const p.max
    , "minY": const p.minY
    , "maxY": const p.maxY
    , "feature": f'
    , "toSVG": unsafePerformEff <<< \_ -> SVG.renderGlyph g
    }
  where p = unwrap $ glyphToGlyphPosition g
        f' = toNullable $
               (\(Feature chr min max _) -> {chr, min, max}) <$> f
```

Note the use of ‘const‘ to produce the constant functions that describe the bounding box, after converting the ‘Glyph‘ to a ‘GlyphPosition‘, and ‘unsafePerformEff‘ to create functions that use the canvas and SVG interpreters to produce the output expected by BD. Since the ‘feature‘ field is optional, ‘toNullable‘ is used to transform an eventual ‘Nothing‘ to an actual JS null, before being placed in the record.

A helper function exists for working with ‘Glyphs‘ in the ‘F‘ functor, which is useful when the ‘Glyphs‘ were constructed in the process of parsing externally provided data. In case of failure, we produce a ‘String‘ containing the errors, which is the format expected by BD:

```
writeGlyph :: forall a c r.
              Maybe (Feature c r)
           -> F (Glyph a)
           -> Foreign
writeGlyph f fG = case runExcept fG of
  Left errors ->
    toForeign $ fold
      $ renderForeignError <$> errors
  Right glyph -> writeGlyph' f glyph
```

In short, ‘writeGlyph‘ produces data, including possible errors, in exactly the format expected by BD, while staying type safe.

#+begin$_{\text{comment}}$

6.5.7 Limitations/Performance

TODO inefficient – rendering tens of thousands of glyphs can be slow, each glyph setting its own stroke & fill colors, even if all glyphs look the same NOTE: still pretty fast! 100k in 8 seconds, and (probably?) O(n).

#+end$_{\text{comment}}$

6.5.8 Summary

6.6 Transforming Events

#+begin$_{\text{comment}}$

6.6.1 Notes

not **really** a problem in BD, however there is no checking that the features provided to listeners actually have the data expected by them, leading to a risk of runtime errors and decreased reusability (hardly unique to BD)

would be horrible when working with events from multiple different sources, e.g. BD and Cy.js – would end up with a bunch of nestled if-else statements, searching for non-null properties. and even when you find all the properties you want, there’s no guarantee that

is the BD API also limited in what can be done? well, not really; I certainly won’t be able to do any more than featurelisteners can do (and only barely in a cleaner/more correct way)

#+end$_{\text{comment}}$

6.6.2 Track interactions

When working with connected data, we want to be able to interact with the data in multiple ways, to explore one data set by examining another. In the architecture of GGB, this comes down to sending events between tracks – when clicking on some data point in one track, an event containing information derived from that data point is created, and sent to other tracks that have been configured to react to those kinds of events.

In short, the system consists of four parts:

1. The browser, e.g. BD, producing raw events in response to user interaction, in whatever

format it uses

1. A track source, mapping the raw event to one used by GGB
2. A track sink, consuming GGB events into some callback that performs effects on...
3. Another browser, e.g. Cy.js.

Each part of this system should also be user-configurable, and constructed in such a way as to minimize the risk of callbacks receiving events they cannot process – we want event type safety.

We begin by looking at what events are provided by BD and Cy.js.

6.6.3 Biodalliance

BD provides several facilities for the user to add event handlers, functions that are called when the user interacts with the browser, or the browser performs some action. We are interested in only one, 'addFeatureListener'. This function adds a handler that is called when the user clicks on a feature, i.e. on a data point in a BD track.

It receives several parameters, the DOM MouseEvent that triggered the handler, the BD feature clicked on, the track the click occured in, and an array of other objects; for simplicity's sake, we only look at the feature clicked on. This feature is a JS object, and can contain any information that BD parsed from the raw data, meaning two features from two different tracks can look very different, and

6.6.4 Cytoscape.js

Cy.js has a vast array of potential interactions and event handlers. We will focus on regular click-events, and thus are interested in the 'cy.on("click")' function, which adds on-click event handlers to the elements matching the provided selector. When no selector is provided, this matches all elements, and the handler functions similarly to the one provided to BD's 'addFeatureListener'.

Handlers attached with 'cy.on()' receive the core Cy.js graph instance, the target element (or graph) that caused the event, as well as information of what kind of event it was and when it was triggered. We're mainly interested in the 'target' value, which is similar to the 'feature' argument in BD's handler. Like with BD, this value contains the entire element clicked on; a big and complex JS object which can contain arbitrary data.

Both BD and Cy.js, then, produce events with unordered information of arbitrary complexity – unordered in the sense that knowledge of the data is required to extract information such as genomic position from it. Even though two pieces of data may both contain position information, there is no reason to expect the data to be found in the same place in the respective JS objects, or be of the same format. Even so, we want a

#+begin$_{\text{comment}}$

6.6.5 Type-safe – but compile-time doesn't make sense

My first attempt, ambitious as it was, failed, and was in fact misguided from the beginning – however, it serves to illustrate the goal, and illuminate the path there. This was to represent the types of events as types in Purescript, via Purescript's row types and polymorphic variants from purescript-variant.

Row types make it possible to express extensible records; they are essentially type-level maps from labels to types. For example, a record in Purescript:

##+BEGIN$_{\text{SRC}}$ purescript exRec :: Record (x :: Number, title :: String) exRec = { x: 123.0, title: "hello" } #+END$_{\text{SRC}}$

Row types can also be open, making it possible to write functions that work with any record containing at least some given fields. Here is a function that works on any record with a field named 'title' of type String:

##+BEGIN$_{\text{SRC}}$ purescript – { label :: Type } is sugar for Record (label :: Type) exRec2 :: { title :: String } exRec2 = { title: "another record" }

titleLength :: forall r. { title :: String | r } -> Int titleLength { title } = length title

titleLength exRec `= 5 titleLength exRec2 =` 14 #+END$_{\text{SRC}}$

variants

The use of row types is not limited to records. The package purescript-variant provides an implementation of polymorphic variants using row types; they are to sum types what records are to product types. For example, this function 'eitherOr' works with all possible Variants, with a default implementation for labels other than "either" and "or". A variant with the label "either" must contain a Boolean.

##+BEGIN$_{\text{SRC}}$ purescript _either = SProxy :: SProxy "either" _or = SProxy :: SProxy "or" _nope = SProxy :: SProxy "nope"

eitherOr :: forall r. Variant (either :: Boolean, or :: Unit | r) -> String eitherOr = default "neither!"

vEither :: Variant (either :: Boolean) vEither = inj _either true

vOr :: Variant (or :: Unit) vOr = inj _or unit

vNope :: Variant (nope :: Maybe Int) vNope = inj _nope (Just 543)

eitherOr vEither `= "either true" eitherOr vOr =` "or unit" eitherOr vNope == "neither!" #+END$_{\text{SRC}}$

The goal of using variants and rows was to provide type-safety of events. An Event would simply be a variant, and the different types of events would have different labels, and thus also different types. Producers and consumers of events would have their own rows to keep track of what they could produce and consume; as a corollary, Purescript's type checker would ensure that a consumer only receives events that it knows how to consume. In other words, a consumer could be connected to a producer if the producer's row is a subset of the consumer's row.

This is all well and good, and my early attempts worked well. Problems arose when attempting to move from a hardcoded event flow to configuring one – this is when I realized that it doesn't make sense to have the compiler check something that needs to be configured by the user, and thus checked at runtime!

as done in Functional Pearl: implicit configurations `http://www.cs.rutgers.edu/~ccshan/prepose/prepose.pdf`)

What I actually desired was a way to express events in an easy to configure way, while also guaranteeing correctness as far as possible, with good error

reporting picking up the slack where necessary.

#+end$_{\text{comment}}$

6.6.6 JSON Configuration and Semantics

What was needed was using a single type for all the possible events, but also providing enough data to do some kind of validation – validation on data coming from and going to the FFI, meaning it cannot be trusted whatsoever.

Since ease of configuration was another important factor, I decided to start there. JSON was the natural format to use for configuration; upon reflection, it also turned out to be a good type for events in GGB.

Having decided on JSON as the configuration format still leaves the question: what does configuring an event entail? We want the user to be able to describe what the events that come from some track look like and contain, as well as describe how the raw events are transformed into GGB events.

In most cases, this focus on the configuration format, versus the actual semantics of what the configuration data will provide, would be a sign of something being quite wrong – the format is an implementation detail.

However, in this case the format and semantics overlap. If an Event is JSON, and the configuration is given in JSON, why not use the Event as configuration? That was the inspiration that led to the current system. The user configures the event system by providing templates, or patterns, that map between raw events and the various events a track produces and consumes. It can be seen as a kind of pattern matching.

6.6.7 TrackSource and TrackSink

The solution consists of two types, ‘TrackSource‘ and ‘TrackSink‘. The former transforms events from browsers to GGB events, the latter handles received events on browser tracks.

TrackSource

The definition in PS of a ‘TrackSource‘ can be seen in code block 6.6.7. It is simply a newtype wrapper over the type of arrays of JSON parsers to some given type ‘a‘. A ‘Functor‘ instance is derived so that we can apply functions to the output of a ‘TrackSource‘, and ‘Semigroup‘ and ‘Monoid‘ instances are defined so that multiple ‘TrackSource a‘s can be combined into one.

```
newtype TrackSource a = TrackSource (Array (Json -> Maybe a))

derive instance functorTrackSource :: Functor TrackSource

instance semigroupTrackSource :: Semigroup (TrackSource a) where
  append (TrackSource s1) (TrackSource s2) = TrackSource (s1 <> s2)

instance monoidTrackSource :: Monoid (TrackSource a) where
  mempty = TrackSource mempty
```

A ‘TrackSource‘ can be constructed by providing a parsing function. However, we want to let the user configure track sources, and not have to write them in PS. The configuration needed for a ‘TrackSource‘ is a name, the JSON structure for the event to be produced, and the JSON structure of the event produced by the underlying track (e.g. Biodalliance).

For parsing all this JSON, the purescript-argonaut library was used, which the next section describes.

Json decoding with Argonaut Argonaut is a library for working with JSON in Purescript, including serializing and deserializing, as well as working with the JSON trees.

One key difference to purescript-foreign and its Foreign type, Argonaut's Json type only corresponds to actual JSON, i.e. things that are legal in JSON formatted files. Thus, functions and other values that cannot be serialized to JSON, cannot be represented in the Json type.

Values of type Json can be decoded, or parsed, in several ways. In this case we're interested in walking arbitrary JSON trees and transforming lists of paths. Before looking at how the parsing works, here is an example of a legal SourceConfig:

```
{
  "eventName": "range",
  "eventTemplate": { "chr": "Chr",
                     "minPos": "Bp",
                     "maxPos": "Bp"
  },
  "rawTemplate": { "segment": "chr",
                   "min": "minPos",
                   "max": "maxPos"
  }
}
```

This defines a source that parses objects/events like this one, the JS object passed to the event handler when clicking on a feature in BD:

```
{
  // ...
  segment: "chr11",
  min: 1241230,
  max: 1270230
  // ..
}
```

Into a JS object that looks like

```
{
  chr: "chr11",
  minPos: 1241230,
  maxPos: 1270230
}
```

This is useful if several tracks produce events with the same data but in objects that look different; the consumer of the event will only see events of this last format. The templates provided can be of arbitrary depth and complexity; the only rule is that each leaf is a key, and all properties be strings (i.e. no arrays). There is some validation too, detailed later.

'eventTemplate' and 'rawTemplate' are both whole structures which we're interested in. For each leaf in the eventTemplate (including its property name), we create a path to where the corresponding value will be placed in the finished event. Similarly, we need to grab the path to each leaf in the rawTemplate, so we know how to grab the value we need in the finished event, from the provided raw event.

Fortunately, Argonaut provides functions for dealing with exactly this. First, the JCursor type describes a path to a point in a JSON tree:

```
data JCursor =
  JIndex Int JCursor
  JField String JCursor
  JCursorTop
```

It can be seen as a list of accessors. If we have an object in JS:

```
let thing = { x: [{a: 0},
                  {b: {c: true}}
                            ]};
```

We can grab the value at 'c' with

```
let cIs = thing.x[1].b.c;
```

With JCursor, this accessor chain 'x[1].b.c' would look like:

```
(JField "x"
   (JIndex 1
    (JField "b"
       (JField "c" JCursorTop))))
```

[1]DEFINITION NOT FOUND.

It's not pretty when printed like this, but fortunately not much direct manipulation will be needed. We create these JCursors from a JSON structure like the templates above with the function toPrims, seen in 6.6.7.

```
toPrims :: Json -> List (Tuple JCursor JsonPrim)
```

The type JsonPrim can be viewed as exactly what it sounds like – it represents the legal JSON primitives: null, booleans, numbers, strings. In this case we only care that they are strings.

This function walks through a given JSON object, and produces a list of each leaf paired to the JCursor describing how to get to it. That is, it does exactly what we want to do with the rawTemplate from earlier.

With the eventTemplate we don't want to pick out the leaf, but the label of the leaf. In this case we do need to step into the JCursor structure, but only a single step, after reversing it:

```
insideOut :: JCursor -> JCursor

eventName <- case insideOut cursor of
               JField s _ -> Just s
               _          -> Nothing
```

The function 'insideOut' does what expected and reverses the path through the tree. We then match on the now first label, and save it as the name. If it was an array, we fail with a Nothing.

Argonaut, especially the functions concerning JCursor, largely uses the Maybe type. This is fine for the most part, but as this will be used in configuration, and thus needs to tell the user what has gone wrong if the provided configuration is faulty, it's not enough.

A more appropriate type would be Either String, which allows for failure to come with an error message. To "lift" the functions using Maybe into Either String. See `source code` for an example.

To provide the user with additional help when configuring, the source configurations are validated to make sure the given JSON structures are legal, or "match". Given some value that we want to have in the finished event, and all of the values we know we can get from the raw event, if we can't find the first value among the latter, something's wrong.

The implementation is simple. The Cursors here are grabbed from the result of toPrims above; the JCursors themselves are unaltered.

```
-- Labelled version of Tuple JCursor String
type Cursor = { cursor :: JCursor
              , name :: String
              }

type RawCursor = Cursor
type ValueCursor = Cursor


validateTemplate :: Array RawCursor
                 -> ValueCursor
                 -> Either String ValueCursor
validateTemplate rcs vc =
  if any (\rc -> vc.name == rc.name) rcs
  then pure vc
  else throwError $ "Event property "
                 <> vc.name
                 <> " is not in raw template"
```

In words, if one of the many raw event cursors has the same name as the given value cursor, it's good, otherwise throw an error. To increase this to validate the array of cursors defining a finished event, we can make use of Either's Applicative instance, and traverse:

```
-- specialized to Either String and Array
traverse :: forall a b.
            (a -> Either String b)
         -> Array a
         -> Either String (Array b)



validateTemplates :: Array RawCursor
                  -> Array ValueCursor
                  -> Either String (Array ValueCursor)
validateTemplates rcs = traverse (validateTemplate rcs)
```

The function tries to validate all given templates, and returns the first failure if there are any. Validation of a collection of things for free!

TrackSink

TrackSinks are configured by providing an event name and a callback. On the PS side, these are type-safe, but there is no way to ensure that functions passed from Javascript to Purescript are type-safe. BD and Cy.js TrackSinks, respectively, should have the following types:

```
newtype TrackSink a = TrackSink (StrMap (Json -> a))

type BDTrackSink = TrackSink (Biodalliance -> Eff Unit)
type CyTrackSink = TrackSink (Cytoscape -> Eff Unit)
```

These are the "expanded" types, for clarity. Note that they are extremely similar; the only difference is what type of browser they work on:

```
BDTrackSink = TrackSink
  (StrMap (Json -> Biodalliance -> Eff Unit)

CyTrackSink = TrackSink
  (StrMap (Json -> Cytoscape    -> Eff Unit)
```

The event name is used to place the function in the correct index of the StrMap. The callback uses currying to take both the event (as JSON) and the respective browser instance, to be used e.g. when scrolling the Biodalliance view to an event.

In 6.6.7 a BD TrackSink is defined that scrolls the BD viewport upon receiving an event.

```
var bdConsumeLoc = function(json) {
    return function(bd) {
        return function() {
            bd.setLocation(
              json.chr,
              json.pos - 1000000.0,
              json.pos + 1000000.0);
        };
    };
};

var bdTrackSinkConfig =
  [ { eventName: "location",
      eventFun: bdConsumeLoc } ];
```

Running TrackSources and TrackSinks

For TrackSource and TrackSink to be usable we need to be able to create them from the provided configurations, and provide functions for applying them to events as appropriate.

TrackSource To create a TrackSource, the provided templates are parsed and validated. Since a TrackSource is a list of parsers, if the SourceConfig is correct,

a function from raw events to parsed events is returned, wrapped in a list and the TrackSource type, as seen in 6.6.7.

```
makeTrackSource :: SourceConfig
                -> Either String (TrackSource Event)
makeTrackSource sc = do
  rawTemplates <- parseRawTemplateConfig sc.rawTemplate
  eventTemplates <- validateTemplates rawTemplates
                    =<< parseTemplateConfig sc.eventTemplate

  pure $ TrackSource $ singleton $ \rawEvent -> do
    vals <- parseRawEvent rawTemplates rawEvent
    evData <- fillTemplate eventTemplates vals
    pure $ { name: sc.eventName, evData }
```

To extend the above function to work on a collection of configuration objects, function composition is used in 6.6.7 to first attempt to use each provided configuration to create a TrackSource, followed by combining the list of parsers into a single one.

```
makeTrackSources :: Array SourceConfig
                 -> Either String (TrackSource Event)
makeTrackSources =
  map fold <<< traverse makeTrackSource
```

First ‘traverse‘ is used to try to create the TrackSources, which returns an array of ‘TrackSource Event‘ if all were legal, or an error if something went wrong. Next, ‘map‘ is used to apply a function to the ‘Right‘ side of the ‘Either‘ from the use of ‘traverse‘, and the applied function is ‘fold‘, which concatenates a collection of values of some monoid into a single value – the monoid in question is TrackSource.

This is not the only reasonable way of defining this function – one may very well want to collect the error messages while returning the successes. As ‘makeTrackSources‘ demonstrates, not much code is needed to compose functions to provide the validation logic that is desired, and there is nothing unique about this function; all that is required is swapping out some of the functions.

Finally, a way to use a TrackSource, to parse a raw event, is required. Code block 6.6.7 shows the function that does so.

```
runTrackSource :: TrackSource Event
               -> Json
               -> Array Event
runTrackSource (TrackSource ts) raw =
  filterMap (_ $ raw) ts
```

It works by applying each function in the array wrapped by TrackSource to the provided value, filtering out the ‘Nothing‘s and returning an array of successfully parsed ‘Events‘.

TrackSink A TrackSink is a map from event names to a function that handles the event, so to make one we create a singleton map from the provided event name to the provided function, and wrap it in the TrackSink type:

```
makeTrackSink :: SinkConfig
              ~> TrackSink
makeTrackSink sc =
  TrackSink
    $ StrMap.singleton sc.eventName sc.eventFun
```

Using a collection of ‘SinkConfigs‘ to produce a single ‘TrackSink‘ is not in itself complicated; see the code is in block 6.6.7. The bulk of the logic is in validation, namely ensuring that there are not multiple handlers for a given event:

```
makeTrackSinks :: forall a.
                  Array (SinkConfig a)
               -> Either String (TrackSink a)
makeTrackSinks scs = do
  let count =
        StrMap.fromFoldableWith (+)
          $ map (\c -> Tuple c.eventName 1) scs

      overlapping =
        StrMap.filter (_ > 1) count

  when (not StrMap.isEmpty overlapping)
    let error = foldMap (append "\n" <<< show)
                  $ StrMap.keys overlapping
    in throwError $ "Overlapping tracksinks!\n" <> error

  pure $ foldMap makeTrackSink scs
```

In this case, we use ‘foldMap‘ to map the ‘makeTrackSink‘ function over the provided configurations, and then use the ‘TrackSink‘ monoid instance to combine them – similar to ‘fold «< traverse‘ in the case of ‘TrackSource‘.

To use a ‘TrackSink‘, we see if a handler for the provided event exists. If it does, we apply it to the contents of the event. In code block 6.6.7 this is done using PS’s ‘do-notation‘ syntax.

```
runTrackSink :: forall a.
                TrackSink a
             -> Event
             -> Maybe a
runTrackSink (TrackSink sink) event = do
  f <- StrMap.lookup event.name sink
  pure $ f event.evData
```

However, since ‘TrackSinks‘ are intended to perform effects, a helper function for that is useful. In particular, the function ‘forkTrackSink‘ in 6.6.7 creates a "thread" that reads events from a provided ‘BusRW‘, running effectful functions from the provided ‘TrackSink‘ if the received event has a handler:

```
forkTrackSink :: forall env.
                 TrackSink (env -> Eff Unit)
              -> env
              -> BusRW Event
              -> Aff Canceler
forkTrackSink sink env bus =
  forkAff $ forever do
    event <- Bus.read bus

    case runTrackSink sink event of
      Nothing -> pure unit
      Just f  -> liftEff $ f env
```

6.6.8 Summary

6.7 User Interface

The main function of GGB's user interface is to tie the browser tracks – BD and Cy.js – together. It also creates and to some extent manages the JS browser track instances, and renders the HTML for the entire UI.

6.7.1 Biodalliance

Biodalliance has a full-featured UI for exploring genomic data chromosome-wise, adding and removing currently displayed tracks, configuring browser options, and exporting the current browser view to various formats. BD accomplishes this by creating and working with DOM elements and the HTML5 canvas API, and setting handlers on DOM events such as clicking and dragging the browser view, or pressing the arrow keys to scroll.

Because BD does not use any abstracting library for dealing with the DOM, and likely because BD has grown features organically over time, the code for updating the UI is interleaved with other code, including event handlers, fetching data for a track, and more. BD also programmatically sets various CSS

properties on UI elements, and uses the web browser's computed stylesheet to figure what manipulations are necessary.

In short, BD's UI uses plenty of global state, and is highly complex and spread out over the codebase. Adding a UI element would require finding a place in the DOM where it would fit – both in screen estate as well as in styling – and somehow suture it into the code while making sure that the existing UI elements are not negatively affected by this sudden new element, plus that the other UI elements and functionality do not interact with the element in some undesired manner.

Another problem, that could arise when adding some feature, not necessarily modifying the UI itself, is the risk of the interface ending up in an inconsistent state. With all the global state that is used, both in the DOM and in the BD browser itself, it is difficult to know what changes can be made. One cannot even call a function which performs some action when a button is clicked, without risking that the function itself toggles some state.

In Purescript, we do not juggle DOM elements and events. Instead, we use Halogen, from purescript-halogen, a type-safe UI library, similar in purpose to React. Event passing between tracks is taken care of by purescript-aff-bus and threads from purescript-aff, while DOM events are handled by Halogen.

Using these tools, we can construct a potentially complex UI, with some, albeit not absolute, confidence that the UI will not move to an inconsistent state. Halogen also provides a DSL for declaratively constructing the DOM of our application. Naturally, there is no implicit global state to be concerned about.

6.7.2 Quick Halogen intro

Halogen is a component-based UI library, using a virtual DOM implementation to render HTML to the browser. A component is a value of the (fairly complicated) type Component (removed constraints etc. for clarity):

```
component ::
  Component    -- (1)
    renderer   -- (2)
    query      -- (3)
    state      -- (4)
    message    -- (5)
    monad      -- (6)
```

The type 'Component (1)' takes five type parameters. The first, 'renderer (2)' is the type used to render the page, we use a HTML renderer. Next is 'query (3)', which is filled with our query algebra, to be explained later; in short it is the set of commands the component can respond to. 'state (4)' is the type of the state kept by the component. We don't have any, so we set it to 'Unit'. 'message (5)' is the type of messages we can produce, which we can send to other parts of the program. Finally, 'monad (6)' is the type in which all effects produced by the component will happen. In our case, it's the 'Aff' monad for asynchronous effects – it could also be a monad transformer stack, or some free monad.

Query algebras

The "Query algebra" is the type describing the possible actions we can query the component to perform. The type is not complicated; in GGB we have:

```
data Query a
  = CreateBD (forall eff.
              HTMLElement
           -> Eff (bd :: BD | eff)
              Biodalliance) a
  | PropagateMessage Message a
  | BDScroll Bp a
  | BDJump Chr Bp Bp a
  | CreateCy String a
  | ResetCy a
```

From top to bottom, we can ask it to ‘CreateBD‘, providing a function that creates a Biodalliance instance given a HTML element to place it in; we can propagate messages from the child components; we can scroll and jump the BD instance; and we can create and reset the Cy.js instance. That's what the queries look like, but we also need to define an ‘eval‘ function. This maps Query to Halogen commands, which are also defined by a functor type – the function is a natural transformation from our Query DSL to the Halogen DSL (a free monad).

```
eval :: Query
     ~> HalogenM
          state
          query
          childQuery
          childSlot
          message
          monad
```

The type parameters of ‘HalogenM‘ are the same as those of ‘Component‘, adding a ‘childQuery‘ type, the Query type of values which this component can use to communicate with its children, and ‘childSlot‘, the type which is used to index into the child components. For the main GGB component they are:

```
type ChildSlot = Either2 UIBD.Slot UICy.Slot

type ChildQuery = Coproduct2 UIBD.Query UICy.Query
```

‘ChildSlot‘ is a coproduct of the two child Slot *types* (Either2) of the child components; we can query the BD slot or the Cy.js slot at once, not both. ‘Either2‘ is a generalization of ‘Either‘ to a variable number of types, a convenience

that makes it easy to change the number of slots, without more work than a type synonym. ‘ChildQuery‘ is a coproduct of the two child Query *functors* (Coproduct2).

```
data Either a b =
  Left a | Right b

data Coproduct f g a =
  Coproduct (Either (f a) (g a))
-- can be viewed as (pseudocode):
data Coproduct f g a =
  Coproduct (Left (f a)) | (Right (g a))

type ChildQuery a =
  Either (UIBD.Query a) (UICy.Query a)
```

We can’t use normal ‘Either‘ for ChildQuery, as we wouldn’t be able to be parametric over the ‘a‘ type in both child queries. If we were to map a function ‘UICy.Query (a -> b)‘ on the Right component of the Either ChildQuery, we’d end up with the type ‘Either (UIBD.Query a) (UICy.Query b)‘, which obviously is not congruent to ‘ChildQuery a‘.

Writing the function is simple enough. We pattern match on the input Query, and produce effects in the HalogenM type. Creating BD is done by querying the BD child using its respective slot and a ChildPath – a type describing a path to the child component, and providing an action to tell the child component to perform.

```
eval = case _ of
  CreateBD bd next -> do
    _ <- H.query'
           CP.cp1
           UIBD.Slot
           $ H.action (UIBD.Initialize bd)
    pure next
```

‘H.action‘ is a Halogen function mapping ChildQuery constructors to concrete actions, by simply applying the ‘Unit‘ type to it.

```
type Action f = Unit -> f Unit
action :: forall f. Action f -> f Unit
action f = f unit
```

Finally, we return the next command. Next is ‘PropagateMessage‘, which receives a Message (sent from the function handling messages from the children):

```
data Message
  = BDInstance Biodalliance
  | CyInstance Cytoscape
```

Depending on which message it is, we print a log message, and then use ‘H.raise‘ to send the message out from Halogen to subscribers elsewhere in the app (more on that later).

```
PropagateMessage msg next -> do
  case msg of
    BDInstance _ -> liftEff $ log "propagating BD"
    CyInstance _ -> liftEff $ log "propagating Cy"
  H.raise msg
  pure next
```

The rest are simple queries to the respective child component, practically the same as ‘CreateBD‘:

```
BDScroll dist next -> do
  _ <- H.query' CP.cp1 UIBD.Slot $ H.action (UIBD.Scroll dist)
  pure next
BDJump chr xl xr next -> do
  _ <- H.query' CP.cp1 UIBD.Slot $ H.action (UIBD.Jump chr xl xr)
  pure next

CreateCy div next -> do
  _ <- H.query' CP.cp2 UICy.Slot $ H.action (UICy.Initialize div)
  pure next
ResetCy next -> do
  _ <- H.query' CP.cp2 UICy.Slot $ H.action UICy.Reset
  pure next
```

Rendering

Next is rendering the component. This is done by providing a function from the component ‘state‘ to a description of the DSL used by the ‘renderer‘ type. In our case, we render to ‘HTML‘, and so use the type ‘ParentHTML‘, which contains all the types required to interact with the children.

```
render :: State
       -> ParentHTML
            query
            childQuery
            childSlot
            m
```

The function itself is simple, we use Arrays and some functions to describe the HTML tree, a simplified version follows:

```
render _ =
  HH.div_
    [ HH.button
      [  HE.onClick
           $ HE.input_
             $ BDScroll
               $ Bp (-1000000.0) ]
      [ HH.text "Scroll left 1MBp" ]

    , HH.div
      [] [HH.slot'
            CP.cp1
            UIBD.Slot
            UIBD.component
            unit
            handleBDMessage]
    ]
```

This produces a button with the text "Scroll left 1MBp", and clicking on it sends a query to ‘eval‘ to scroll the BD view 1 MBp to the left; as well as a div with the BD child component. Adding the child component here is how we create the component, so we must also provide a handler in the parent for messages from the child, namely ‘handleBDMessage‘.

Messages

A component can send messages to its parent, or the rest of the application in the case of the top-level component. These are the messages the BD and Cy.js components can produce, respectively:

```
data UIBD.Message
  = SendBD Biodalliance

data UICy.Output
  = SendCy Cytoscape
```

The main component can produce these:

```
data Message
  = BDInstance Biodalliance
  | CyInstance Cytoscape
```

Note that the main container uses its own messages to propagate the children components; message passing is limited by Halogen, and anything more complex than this should be done on another channel (which is what GGB does with events).

The messages from the BD and Cy.js components are handled by the functions ‘handleBDMessage‘ and ‘handleCyMessage‘:

```
handleBDMessage :: UIBD.Message
                -> Maybe (Query Unit)
handleBDMessage (UIBD.SendBD bd) =
  Just $ H.action $ PropagateMessage (BDInstance bd)

handleCyMessage :: UICy.Output
                -> Maybe (Query Unit)
handleCyMessage (UICy.SendCy cy) =
  Just $ H.action $ PropagateMessage (CyInstance cy)
```

Note that these produce Queries on the main component. We want to send the messages containing the references to the instances out from the component to the outside application, hence creating a PropagateMessage query wrapping the reference. As seen in ‘eval‘ above, this in turn callse ‘H.raise‘ on the message, sending it to the outside world.

Creating the component

These functions, including one to produce the initial state (simply ‘const unit‘) are all put together and provided to the ‘parentComponent‘ function, producing the Component itself. This can then be provided to Halogen’s ‘runUI‘ function, along with the initial state and an HTML element to be placed in, to create and run the Halogen component.

First, however, we need a ‘main‘ function application to run.

6.7.3 The main application

‘main‘ is the function which will be called by the user to run the browser. It takes a ‘Foreign‘ object – the one to parse into a browser configuration – and then does some stuff with Eff (e.g. be a genetics browser):

```
main :: Foreign -> Eff _ Unit
main fConfig = HA.runHalogenAff do
```

First we attempt to parse the provided configuration, logging all errors to config on failure, otherwise continuing:

```
case runExcept $ parseBrowserConfig fConfig of
  Left e -> liftEff $ do
    log "Invalid browser configuration:"
    sequence_ $ log <<< renderForeignError <$> e

  Right (BrowserConfig config) -> do
```

With a validated config, we can create the track/graph configs, and create the function which will later be used to create Biodalliance:

```
let {bdTracks, cyGraphs} = validateConfigs config.tracks

    opts' = sources := bdTracks.results <>
            renderers := config.bdRenderers

liftEff $ log $ "BDTrack errors: " <> foldMap ((<>) ", ") bdTracks.errors
liftEff $ log $ "CyGraph errors: " <> foldMap ((<>) ", ") cyGraphs.errors

let mkBd :: (forall eff. HTMLElement -> Eff (bd :: BD | eff) Biodalliance)
    mkBd = initBD opts' config.wrapRenderer config.browser
```

After picking the element to run in, we create the Halogen component, and create the Buses to be used by the events system. Note that we bind the value of ‘runUI‘ to ‘io‘:

```
io <- runUI component unit el'

busFromBD <- Bus.make
busFromCy <- Bus.make
```

‘io‘ can be used to subscribe to messages sent from the main component, as well as send queries to it, which we do momentarily. First, we use the provided TrackSink and TrackSource configurations to create the BD TrackSink and TrackSource:

```
let bdTrackSink =   makeTrackSinks <<< _.bdEventSinks =<<
                      note "No BD event sinks configured" (config.events)
    bdTrackSource = makeTrackSources <<< _.bdEventSources =<<
                      note "No BD event sources configured" (config.events)
```

We create the respective values, adding an error message if something went wrong.

Finally, we attach a callback to the Halogen component to listen for the reference to the BD instance, sent by the BD component upon creation. We

then use the TrackSink and TrackSource configurations to hook BD up to the event system. Finally, we ask the main component to create the BD instance:

```
io.subscribe $ CR.consumer $ case _ of
  BDInstance bd -> do

    case bdTrackSink of
      Left err -> liftEff $ log "No BD TrackSink!"
      Right ts -> forkTrackSink ts bd busFromCy *> pure unit

    liftEff $ case bdTrackSource of
      Left err -> log err
      Right ts -> subscribeBDEvents ts bd busFromBD

      --TODO remove BDRef? debug stuff...
    liftEff $ setBDRef bd
    pure Nothing

  _ -> pure $ Just unit

io.query $ H.action (CreateBD mkBd)
```

If the ‘TrackSink‘ was correctly configured, ‘forkTrackSink‘ is used to pipe events from the Cytoscape.js instance to the handler defined by said ‘TrackSink‘. We don’t care about being able to kill the "thread" using the ‘Canceler‘, so we throw away the result with ‘*> pure unit‘. Similarly, the ‘TrackSource‘ is used with the helper function ‘subscribeBDEvents‘, defined thusly:

```
subscribeBDEvents :: forall r.
                     (TrackSource Event)
                  -> Biodalliance
                  -> BusRW Event
                  -> Eff _ Unit
subscribeBDEvents h bd bus =
  Biodalliance.addFeatureListener bd $ \obj -> do
    let evs = runTrackSource h (unwrap obj)
    traverse_ (\x -> Aff.launchAff $ Bus.write x bus) evs
```

It adds an event listener to the provided BD browser instance and writes the successful parses to the provided Bus.

The Cytoscape.js code is analogous.

6.8 Summary

Many (actual and potential) problems related to legacy code were encountered during development, and they were dealt with in various ways using different

features of the language and libraries provided. The resulting application is written in a purely functional language, but also both embeds a legacy system within it, as well as providing tools for extending said legacy system. Last but not least, it is an extensible foundation for a new genome browser. In the Discussion that follows, some more general ways that pure FP had an impact on this project are considered, and an effort is made to extrapolate these results to other legacy systems.

Chapter 7

Discussion

The resulting browser, GGB, fills the specified requirements, and the codebase does as well. Purescript was not a magic bullet, and some problems were experienced during development, however the source code on the whole does not, arguably, suffer from the problems of legacy code as defined in this report.

The code from the previous chapter is examined to see if and how it stuck to the ideas of "good" code, including what FP features contributed. Problems experienced during development are presented, and some examples of how GGB will be extended in the future are given, together with estimates on how "easy" the addition of those parts to the codebase and system will be. The potential ways to extend the browser are also looked at from the view of BD, to examine how difficult it would be to extend BD without the support of GGB.

7.1 PureScript for Working with Legacy JavaScript

PureScript was an excellent tool for developing GGB; in the author's opinion, it was no doubt the best choice. PS's FFI capabilities made it easy to wrap the native JS APIs of BD and Cy.js, then construct type-safe PS APIs. This minimizes the reach of JS code into the GGB codebase. Several times during development, when some part of the system started behaving strangely, it was due to the FFI almost every time. Even JS code, which due to its untyped dynamic nature tends to lead to bugs in unexpected places, becomes easy to debug when restricted to the very edges of the program.

However, it is possible that BD was uncharacteristically simple to hook into with another application. While the JS API that is exposed by BD is minimalistic, the "incisions" that had to be made into BD's source code to make it possible for GGB to add some given functionality, were few and quite small.

As an example, adding support for external renderers (something done before this project as a part of Google Summer of Code 2016[1]), while a large undertaking in terms of the time required to understand what had to be done, and rewriting functions to better understand the rendering system as a whole, very few changes were made to the control flow of the rendering system. That is,

[1]For more details, see the author's blog post: `https://chfi.se/posts/2016-07-26-gsoc-render-complete.html`

few, if any, changes that may have led to undesired changes of system behavior, were required.

On the GGB side, this can be seen in the fact that the entire ‘Glyph‘ system of defining new ways of displaying data is mapped to BD by one relatively small function, seen in code block 6.5.6.

‘Glyph‘ is also an excellent argument for PS and GGB being extensible and easy to maintain. For example, if a more precise way of checking whether a glyph covers a given point on the canvas (something currently done by providing axis-aligned bounding boxes) is desired, it is simple to create a new interpreter that maps glyphs to some other more exact collision shape.

Another potential change that is easily facilitated by ‘Glyph‘’s construction using the ‘Free‘ monad would be to add support for new ways of rendering the browser, e.g. using WebGL. Write another interpreter that maps glyphs to whatever the rendering system requires, and suddenly all existing glyphs (and, by extension, all functions that e.g. work on collections of glyphs) are compatible with this new rendering system.

In short, embedding a legacy JS app in a PS app was in this case an efficient way of adding achieving the developmental goals of the project, as interfacing with BD required little overhead. It helped that work had already been done in that area; other projects may or may not present difficulties for the approach taken here.

7.2 Purely Functional Programming and Code Quality

Recall that our definition of "good code" is code which is easy to understand when looked at. Our heuristics for good code include functions that do not have far-reaching or otherwise difficult to predict side-effects; conversely, it is also good to be able to see at a glance what the purpose of some given function is.

Let us first examine how pure FP has led to functions and code that more clearly presents their purpose.

Configuration in GGB is done by parsing a foreign JS object into a record; this record is then used to initialize the browser itself.

Code reuse is increased by features such as typeclasses and parametric polymorphism. The use of a ‘Free‘ monad DSL in the ‘Glyph‘ parts is a good example of this.

Fewer logic problems thanks to representing values in types that correspond to their actual units as semantically relevant to the program behavior, rather than their runtime representation.

Abstractions such as ‘Semigroup‘, ‘Functor‘, etc. provide the programmer with information about what a function does. Code blocks 10 and 11 show the type signatures for two functions, ‘mystery1‘ and ‘mystery2‘. The function implementations are in fact the same, but the information given in the types is not.

If we look only at ‘mystery1‘, the type give us some idea of what the function does, but not much. Does it return the first list in the array? The last? Maybe it creates a list containing the maximum value of each list in the array. Because

the function knows so much about the types it works on, there is a lot that it can do.

On the other hand, see the type of ‘mystery2‘. The only tools at our disposal are those from the ‘Foldable‘ and ‘Monoid‘ typeclasses: from ‘Foldable‘, we know we can walk through the collection, applying a binary function on the current and next value; from ‘Monoid‘, we have a binary operation that combines two of the values in the collection, and we can create an empty value. Code block 12 shows the implementation.

```
mystery1 :: Array (List Int)
         -> List Int
```

Listing 10: A concrete type for some function on an array of lists

```
mystery2 :: forall f m.
            Foldable f
         => Monoid m
         -> f m
         -> m
```

Listing 11: A general type for some function on some foldable of a monoid

```
mystery2 = fold
mystery1 = mystery2
```

Listing 12: The mystery is solved

Next, the impact of pure FP on understanding the consequences of calling functions. One of the reasons that it can be difficult to see what the effects of calling some function in an impure language such as JS is that the function can not only do more or less whatever it wants with whatever it has at its disposal (which is not necessarily easy to discover, either), but it can also call other functions with arbitrary effects, and so on, and so on. The programmer must walk the entire path of function calls to discover what happens.

Pure FP inverts this, as all effects, be it potential failure in a function, or updating the browser DOM, or mutating a mutable array, are encapsulated in the type system. This makes it impossible to call an effectful function from a pure function. The upshot of this inversion is that code cannot cause far-reaching consequences; there is none of the "spooky action at a distance" that is one of the main problems when trying to comprehend legacy code.

7.3 Developmental Difficulties

There was not much of a design specification for GGB at the start, and as the author did not have previous experience with developing a large pure FP application, there were many design and implementation dead-ends on the way to the current state of the product, as it was difficult to decide on the architecture

of the application, and sometimes how to approach and solve smaller problems. Deciding on some abstraction too early was a mistake repeatedly made, leading to refactoring to a more specific implementation. [2] Often, once the problem was solved, a more general solution showed itself, and another refactoring was made. Thanks to the type system, even large refactorings were quite easy, as the compiler largely ensured that the code at the end of the refactoring effort behaved the same as before.

Pure FP is a very different beast from imperative programming, not only when it comes to larger-scale program design. Many of the techniques used in this project require considerably more knowledge to understand than similar solutions in imperative languages. Using typeclass constraints ('Semigroup' etc.) require the programmer to know their meaning, which can become overwhelming when a function uses half a dozen or more constraints. [3] Highly abstract code also tends to become small and terse, which can also be a problem, especially for newcomers to the language and paradigm.

This is not to imply that these abstractions etc. must be used to write good PS code, nor that they are required to gain any of the benefits of pure FP. They can in fact be ignored, while still reaping most of the benefits, as immutability, purity, and type-safety are still provided.

However, it can be difficult to ignore, as one inevitably stumbles upon them when reading library code — something that is likely to occur, as many libraries suffer from having too little documentation. This is a symptom of PS being such a young language, rather than some inherent problem with it. Fortunately this problem is mitigated by PS having a very enthusiastic and welcoming community, filled with people glad to help newcomers with their problems.

7.4 Summary

Pure FP was in many ways a boon in the development of GGB. While there were problems during development, their impact was limited by the support of the type system and purity; nor is it likely that other problems would not have occurred if GGB was developed using in imperative language and idioms.

Other legacy systems are likely to present their own difficulties if one were to attempt a similar project with them, however the opposite may also be true. In the next chapter, other tools that could provide similar benefits are considered, along the extent to which PS was responsible for the benefits in this case. Potential future areas of research are also examined, as is the future of GGB.

[2] Admittedly, the author tended to attempt to use newly learned (but not necessarily understood) concepts and abstractions throughout the project.

[3] On the other hand, one only needs to learn how 'Semigroup' works once.

Chapter 8

Conclusion

Pure FP was a good tool for GGB and BD, however other legacy systems may be more difficult to use the techniques presented in this thesis. For example, it may not be possible to find single points of entry a lá the glyphs in BD's rendering system, or it may simply not be feasible to embed the application due to its function, language, etc.

Even then, it is possible to write pure code, and take advantage of its benefits. Pure functions can be written in practically any language, and they provide the same benefit as shown in this report. Abstractions such as provided by the 'Monoid' and 'Functor' typeclasses can also be represented in other languages. Doing this will lead to code that solves problems in ways much like code written in PS.

Why use a language such as PS, then? What one misses out on by using an impure language is the enforced purity and encapsulated effects; that is, the compilator cannot help as much. Sometimes other tools can be configured to enforce purity in an impure language, e.g. Bodil Stokke's 'cleanjs' configuration for ESLint, a JS linter[1].

However, without a type system akin to that of PS, the compilator simply cannot provide much assistance. That is, one gains the advantages of code that is less likely to have far-reaching and difficult to predict consequences, but it is still up to the programmer to keep the program semantics in their head, and ensure that the program is correct. Conversely, there are fewer tools to assist the programmer in understanding the code and how it relates to the system.

To summarize, pure FP has its own set of problems, however those problems largely concern the knowledge of techniques, idioms, etc., rather than the knowledge of systems developed with it. These can be seen as "up front" problems, which can be solved with developer training. If it is the case that pure FP reduces legacy code problems, this is a much smaller investment than maintaining a legacy system.

[1] 'cleanjs' can be found at its GitHub repository: `https://github.com/bodil/eslint-config-cleans`

8.1 Limitations and Future Work

This project was only that, a single project. It is impossible to draw conclusions that are in any way definite with a sample size of 1; hopefully larger, more quantitative studies are undertaken in the future.

More generally, there has been little research comparing programming languages, and even less research on the user experience of languages, i.e. on what actually makes a good language[20, p. 252]. That is another avenue for future research.

This is an area of research that it is difficult to do studies in, as while there are plenty of legacy systems to go around, maintaining one is expensive enough without experimenting with new approaches, or spending additional money on the overhead of performing studies. When one considers that to get enough data to learn something concrete, experiments such as this one would have to be done on many legacy systems, it seems unlikely that such research will be done.

Another way would be to analyze open source software projects. The data already exists in the form of source control repositories, commit logs, etc. There would be other difficulties, such as controlling for e.g. community culture and other differences between languages used. Nevertheless, some studies have been done in this area recently[18], and the future may be promising.

8.2 Future of the Graph Genetics Browser

GGB will continue to evolve beyond what has been described in this report, including more tools for interactive exploration of genome/graph data, and a genome browser written in PS which would replicate some of the functionality of BD.

Based on the development process thus far, including the fact that several large refactoring efforts have been successfully undertaken, it appears likely that GGB's codebase is flexible enough that adding these features will not be too difficult.

Bibliography

[1] K. Bennett. Legacy systems: coping with success. *IEEE Software*, 12 (1):19–23, jan 1995. doi: 10.1109/52.363157. URL `https://doi.org/10.1109/52.363157`.

[2] J. Bisbal, D. Lawless, Bing Wu, and J. Grimson. Legacy information systems: issues and directions. *IEEE Software*, 16(5):103–111, 1999. doi: 10.1109/52.795108. URL `https://doi.org/10.1109/52.795108`.

[3] Ludovic Courtès. Functional package management with guix. *arXiv preprint arXiv:1305.4584*, 2013.

[4] Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. Nix: A safe and policy-free system for software deployment. In *LISA*, volume 4, pages 79–92, 2004.

[5] Thomas A Down, Matias Piipari, and Tim JP Hubbard. Dalliance: interactive genome viewing on the web. *Bioinformatics*, 27(6):889–890, 2011.

[6] L.H. Etzkorn and C.G. Davis. Automatically identifying reusable OO legacy code. *Computer*, 30(10):66–71, 1997. doi: 10.1109/2.625311. URL `https://doi.org/10.1109/2.625311`.

[7] Michael Feathers. *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[8] Max Franz, Christian T Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D Bader. Cytoscape. js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2):309–311, 2015.

[9] Phil Freeman. Declarative uis are the future — and the future is comonadic! `http://functorial.com/the-future-is-comonadic/main.pdf`, 2017. Accessed: 2018-01-02.

[10] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989. doi: 10.1093/comjnl/32.2.98.

[11] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. doi: 10.1109/PROC.1980.11805.

[12] Valéria Lelli, Arnaud Blouin, Benoit Baudry, Fabien Coulon, and Olivier Beaudoux. Automatic detection of GUI design smells. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems - EICS '16*. ACM Press, 2016. doi: 10.1145/2933242.2933260. URL `https://doi.org/10.1145/2933242.2933260`.

[13] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th international conference on Software engineering - ICSE '97*. ACM Press, 1997. doi: 10.1145/253228.253354. URL https://doi.org/10.1145/253228.253354.

[14] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, may 2006. doi: 10.1007/s10664-006-9002-8. URL https://doi.org/10.1007/s10664-006-9002-8.

[15] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[16] Megan K Mulligan, Khyobeni Mozhui, Pjotr Prins, and Robert W Williams. Genenetwork: A toolbox for systems genetics. *Systems Genetics: Methods and Protocols*, pages 75–120, 2017.

[17] David A. Powner. Federal agencies need to address aging legacy systems. Technical report, U.S. Government Accountability Office, 2016. URL http://www.gao.gov/products/D13445.

[18] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, 2014. doi: 10.1145/2635868.2635922. URL https://doi.org/10.1145/2635868.2635922.

[19] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, 2016. doi: 10.1145/2884781.2884848. URL https://doi.org/10.1145/2884781.2884848.

[20] Peter Seibel. *Coders at work: Reflections on the craft of programming*. Apress, 2009.

[21] Clauirton Siebra, Tatiana Gouveia, Leonardo Sodre, Fabio QB Silva, and Andre LM Santos. The anticipated test design and its use in legacy code refactoring: lessons learned from a real experiment. In *Information Technology for Organizations Development (IT4OD), 2016 International Conference on*, pages 1–6. IEEE, 2016. doi: 10.1109/it4od.2016.7479256. URL https://doi.org/10.1109/it4od.2016.7479256.

[22] Leonardo Humberto Silva, Marco Tulio Valente, Alexandre Bergel, Nicolas Anquetil, and Anne Etien. Identifying classes in legacy JavaScript code. *Journal of Software: Evolution and Process*, 29(8):e1864, apr 2017. doi: 10.1002/smr.1864. URL https://doi.org/10.1002/smr.1864.

[23] GHC Team et al. The glorious glasgow haskell compilation system user's guide, 2005.

[24] A. M. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2(4):153–163, 1937. doi: 10.2307/2268280.

[25] Bruce W. Weide, Wayne D. Heym, and Joseph E. Hollingsworth. Reverse engineering of legacy code exposed. In *Proceedings of the 17th international conference on Software engineering - ICSE '95*. ACM Press, 1995. doi: 10.1145/225014.225045. URL `https://doi.org/10.1145/225014.225045`.

[26] Brent A Yorgey. Monoids: Theme and variations (functional pearl). In *ACM SIGPLAN Notices*, volume 47, pages 105–116. ACM, 2012.