Extending Legacy Software with Functional Programming

Christian Fischer

Christian Fischer
HT 2017
Master's Thesis in Interaction Technology and Design, 30 credits
Supervisor: Anders Broberg
Examiner: Ola Ringdahl

Abstract

Legacy systems are everywhere. Immense resources are placed on fixing problems caused by them, and on legacy system maintenance and reverse engineering. After decades of research, a solution has yet to be found. In this thesis, both the viability of using purely functional programming to mitigate problems of legacy systems is investigated, as well as the possibility that purely functional programming can lead to code that is less likely to lead to legacy problems in the first place. This was done by developing a genome browser in PureScript that embeds, interfaces with, and extends, an existing genome browser written in JavaScript. The resulting codebase is examined, and various characteristics of purely functional programming, and how they helped solve or avoid problems related to legacy systems, are presented. In Conclusion, PureScript is found to be an excellent tool for working with legacy JavaScript, and while the nature of the project limits the conclusions that can be drawn, it appears likely that using purely functional programming, especially with a language such as PureScript that provides a powerful type-system for ensuring program correctness, leads to code that is more easily understandable, and thus avoids the problems of legacy code.

Acknowledgements

This thesis would not have been possible without the support and patience of my supervisor and mentor, Pjotr Prins. I am also grateful for the help of my university supervisor, Anders Broberg, for vital help especially in structuring this report. Last but not least, I would like to thank Professor Rob Williams for providing me with the opportunity to present my work-in-progress at Complex Trait Community in Memphis, TN, June 2017.

Contents

1 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 [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.

1.1 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 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.

1.2 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, with all the limitations inherent to case studies.

1.3 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 GGB 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.

2 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.

2.1 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.

2.1.1 Definition

There is no formal definition of "legacy code", but Feathers 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.
>
> [25]

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. Feathers 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 is not 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 behavior of the system 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.

2.1.2 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.

2.1.3 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 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 compared to the intuitively strange interactions of entangled particles in 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. However, 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 object-oriented programming (OOP) has often been tried as a 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.

2.1.4 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.

2.2 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).

2.2.1 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 are used to express computation.

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, 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.

2.2.2 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.

2.2.3 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.

2.3 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 like a large logical leap 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.

3 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 project is introduced, followed by how the hypothesis was evaluated in the context of this project.

3.1 Functional programming and legacy code

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 theyve 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 — its 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.

3.1.1 PureScript

PureScript is a purely functional programming language in the style of Haskell, that compiles to JavaScript[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.

3.1.2 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.

3.1.3 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 listing 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

[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, so it too is captured by the type system.

[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.

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

3.1.4 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 listing 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 listing 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

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 listing 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

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

Listing 4: Functor typeclass definition

`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 listing 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.

3.1.5 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.

3.2 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.

3.2.1 Biodalliance

Biodalliance (screenshot in figure 1) is an open source (BSD licensed) HTML5-based genome browser written in JavaScript [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.

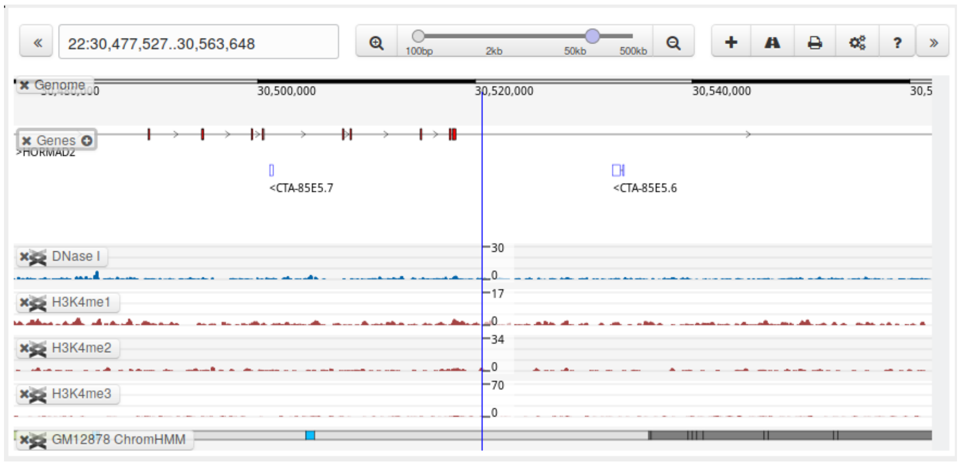

Figure 1: Screenshot of Biodalliance, showing genes and multiple tracks with various phenotypic data.

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) divided over some 61 files[5]. The codebase has grown organically over time, and has become complex, with single pieces of functionality (such as rendering data to the HTML5 canvas) being split into several large functions in different files, and difficult-to-grasp program control flow.

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.

3.2.2 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

[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 at `https://github.com/dasmoth/dalliance`

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, Cy.js (screenshot in figure 2), 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.

3.3 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.

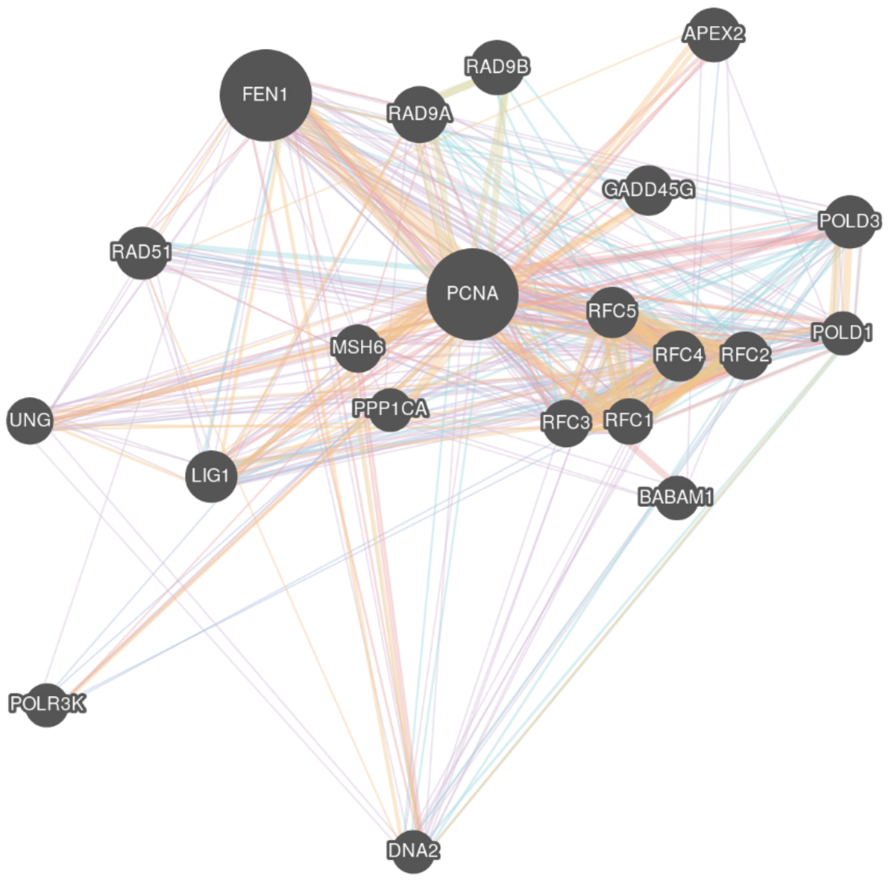

Figure 2: Screenshot of Cytoscape.js, displaying `http://js.cytoscape.org/demos/colajs-graph/`.

4 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 the parts of the source code that are most relevant to the thesis objective.

4.1 The Graph Genetics Browser

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.

The source code for GGB can be found in its GitHub repository at
`https://github.com/chfi/purescript-genetics-browser`

Visually, GGB does not yet add anything substantial on its own; a screenshot is elided as simply placing figure 1 just above figure 2 provides a sufficient idea.

4.2 Translating JavaScript interfaces to PureScript

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).

4.2.1 PureScript's FFI

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. Listing 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.showAppend = function(a) {
    return function(b) {
        return function() {
            console.log(a + b);
        }
    }
}
```

Listing 6: Unsafe function prints the result of "summing" two values, 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 listing 7 limits using the `showAppend` function on strings, and returns an effect [1], making the function pure and behave reasonably.

```
foreign import showAppend
  :: String -> String -> Eff Unit
```

Listing 7: A safe type signature for the function defined in listing 6.

[1]The currently latest version of PureScript, version 0.11.7, uses "effect rows" to annotate what native JS effect |Eff| functions perform. E.g. |showAppend|'s return value would be |forall e. Eff (console :: CONSOLE | e) Unit|, the |console :: CONSOLE| bit signifying that the JS console is used. Effect rows have been removed from the upcoming version of PS, 0.12, and are elided in this thesis, for that reason as well as to reduce space.

The following sections present how the FFI was used to create the modules wrapping the BD and Cy.js APIs.

4.2.2 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 listing 8.

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

Listing 8: The data type representing a BD browser instance.

An FFI function to wrap the BD browser constructor is also required. As seen in listing 9, this takes the browser constructor, another helper function, and the BD configuration as arguments. The output of the function is a continuation that takes an HTML element to place the BD browser in, and returns an effectful function which creates and returns the BD instance.

```
foreign import initBDimpl
  :: Fn3
     Foreign
     RenderWrapper
     BrowserConstructor
     (HTMLElement -> Eff Biodalliance)
```

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

BD can produce events, and for GGB's event system we need to be able to attach a handlers to parse and transmit them. Listing 10 shows newtype that wraps the events from BD, to ensure that raw event are not used in the wrong places, 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 a.
     EffFn2
     Biodalliance
     (BDEvent -> Eff a)
     Unit
```

Listing 10: Type and FFI import for BD events.

In listing 11 the actual foreign function definition is provided.

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

Listing 11: JS implementation of BD event listener function.

This is not the entire BD module, however the other functions are similar. The corresponding Cy.js module follows.

4.2.3 Cytoscape.js

Again, a foreign type for the Cy.js browser instance is required. We also have types for Cy.js elements, collections, and, like BD, a newtype wrapper for events. These types are in listing 12.

```
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
```

Listing 12: Foreign types used in Cytoscape.js interface.

The Cy.js constructor is similar to BD's. Unlike BD, as Cy.js is provided as a dependency to GGB, we can create an instance directly with the imported Cy.js library rather than pass the constructory explicitly as an argument. The constructor also takes an HTML element and an array of JSON objects to be used as the initial graph. Listing 13 shows the type signature for the constructor.

```
cytoscape :: Maybe HTMLElement
          -> Maybe JArray
          -> Eff Cytoscape
```

Listing 13: Type of Cy.js constructor function.

The Cy.js browser instance can be worked with in various ways. Data can be added to the graph, retrieved from it, and deleted, using the functions shown in listing 14.

The graph layout can be controlled with the `runLayout` function, see listing 15, which takes a `Layout` value to update the Cy.js browser's current layout.

`Layout` is a newtype wrapper over `String`, defined as in listing 16, which is what the Cy.js layout function expects. This newtype lets us easily support all the layouts supported by Cy.js, while minimizing the risk of using a string that does not

```
graphAddCollection
  :: Cytoscape
  -> CyCollection Element
  -> Eff Unit

graphGetCollection
  :: Cytoscape
  -> Eff (CyCollection Element)

graphRemoveCollection
  :: CyCollection Element
  -> Eff (CyCollection Element)
```

Listing 14: Types for functions on the Cy.js graph.

```
runLayout :: Cytoscape
          -> Layout
          -> Eff Unit
```

Listing 15: Type of ‘runLayout‘.

correspond to a layout, which would cause an error at runtime.

```
newtype Layout = Layout String
circle = Layout "circle"
```

Listing 16: Layout newtype and example value.

Cy.js produces events in JSON format, like BD. A function to attach event handlers, and a newtype wrapper to keep things safe, are used in GGB; they are analogous to the BD implementations, and so details are elided here.

Unlike BD, the Cy.js API provides a data structure for working with collections of Cy.js elements, and functions on them. Some of these are describe next.

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.

Cy.js provides functions for combining several `CyCollections` in various ways. Listing 17 shows the FFI definition of the function that returns the union of two provided collections, and listing 18 the type signature in the FFI import, taking the opportunity to also define an instance of the Semigroup typeclass on `CyCollection` using `union`.

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

Listing 17: Foreign function wrapping the Cy.js union function on two Cy.js collections.

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

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

Listing 18: FFI import of union and definition of Semigroup instance on CyCollection.

Another common interaction with a collection is extracting a subcollection. With `CyCollection`, we can use the `filter` function for this, as seen in listing 19 (foreign definition elided). The `Predicate` type is another newtype, wrapping functions from the given type to Boolean.

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

Listing 19: Filter on a CyCollection.

The Cytoscape.js API provides some basic predicates on elements, nodes, and edges. See listing 20.

Multiple predicates can easily be combined and manipulated. By composing a predicate on a JSON value with a function that transforms a Cy.js element into JSON, it is easy to create new predicates on Cy.js elements. In addition, `Predicate` is also an instance of the `HeytingAlgebra` typeclass, which generalizes most of the common boolean operations, including disjunction and conjunction. Listing 21 uses these tools to construct complex predicates on Cy.js elements.

The Cy.js API is considerably larger and more complex than that for BD. To ensure correctness beyond what the types provide, the next section briefly describes how a subset of the module is tested.

Tests

PS has a testing framework called `purescript-spec`, which these unit tests are written to use. The `fail` function fails the test with the given string, and the `shouldEqual` function fails if the two arguments are not equal.

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

Listing 20: Imported predicates on Cy.js elements.

```
hasName :: Predicate Json
hasName = Predicate f
  where f json = fromMaybe false
                 $ json ^? _Object <<< ix "name"

  - Composing a JSON-predicate with an element-to-JSON function
elemHasName :: Predicate Element
elemHasName = elementJson >$< hasName

  - Using && and || on Predicates to combine filters
namedNodeOrEdge :: Predicate Element
namedNodeOrEdge = (elemHasName && isNode) || isEdge
```

Listing 21: Combining predicates by composition makes it easy to construct complex filters.

`CyCollection` is unit tested to help ensure that the graph operations work as expected. Listing 22 shows unit tests that provide some assurance that the set operations on `CyCollections` behave as expected. `eles` is a `CyCollection`, `edges` and `nodes` are the corresponding subsets of the collection.

The properties that are tested are, first, that subsets of a collection are, in fact, contained in the collection, and second, if provided the nodes and edges of a collection, the collection itself can be reconstructed.

```
let edges = filter isEdge eles
    nodes = filter isNode eles

    - Signal test failure if these subsets of the graph
    -  are not contained in the graph
when (not $ eles `contains` edges)
       (fail "Graph doesn't contain its edges")

when (not $ eles `contains` nodes)
       (fail "Graph doesn't contain its nodes")

   - The union of the nodes and edges of a graph,
   - should equal the whole graph.
(edges <> nodes) `shouldEqual` eles
(nodes <> edges) `shouldEqual` eles
(edges <> nodes) `shouldEqual` (nodes <> edges)
```

Listing 22: Testing that the edges and nodes of a graph are subsets of the graph, and

4.2.4 Summary

Modules providing subsets of the APIs presented by BD and Cy.js were written using PS's FFI, allowing for some degree of correctness even when working with JS code, with additional safety created using some unit tests in the case of the more complex parts.

The next section describes the configuration system used by GGB, and how it is used together with the modules described in this section to create BD and Cy.js browser instances.

4.3 Safe application configuration

Software needs to be configurable. GGB has many pieces that can and/or need to be configured by the user, such as what data to display. There are also functions that need to be provided from an external source, such as the BD browser constructor.

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 (i.e. a key-value map with strings as keys), and each configuration piece is simply assigned to its respective application variable, large amounts of (boilerplate) code need to be written to validate and verify that the configuration object is correct. Otherwise, there is risk of some subpiece being misconfigured, or simply missing, leading to strange program behavior, or crashes at runtime.

In this section, we examine how configuration is done in BD, and some problems associated with it. Next, the configuration system used in GGB, and how it avoids those problems, is presented. The section ends by showing how the configuration of the embedded BD and Cy.js browsers in GGB works.

4.3.1 Configuring Biodalliance

To give an idea of how configuration can take place in a legacy JS codebase, we look at BD. BD is highly configurable, beyond which tracks to display and how. This information is provided by the user as a JS object, see example in listing 23, which is passed to the browser constructor. The constructor then takes care of configuring the browser.

```
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'
    }]
});
```

Listing 23: Slimmed down BD instance configuration.

The configuration in listing 23 configures some 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 basepairs the browser displays at start); and an array of track source definitions (`sources`), which define what data to show, and how. In this case there is only one track, a mouse genome sequence fetched from the BD website.

There are many more parts of BD that can be customized, and all options are passed in the same object. All such options are provided as JS objects, which are then passed to various functions that e.g. initialize of parts of the browser UI.

Since the options are used as function arguments, the specification of the entire system configuration, including what parts of the configuration object are used, and what values are legal, are spread out over the definitions of all the functions that options are passed to.

Now we take a brief look at some parts of the BD initialization process to get an idea of how the BD configuration object is used.

The Biodalliance initialization process

The initialization of a BD browser instance is highly complex, spread out over many functions and thousands of lines of source code. Here we describe the general methods used to initialize the browser state using the provided configuration.

BD has many features which make use of data stored in the main browser instance. Thus a large part of the initialization process consists of initializing these fields, either by setting them to hardcoded initial values, to values provided by the configuration, or defaults if no option was provided.

BD makes an effort to perform some validation for some of the configuration options. For example, in listing 24 BD ensures that the provided initial position of the browser view is a number. If it is not, BD crashes with an appropriate error. Not that this check requires several lines of code.

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

this.viewStart = opts.viewStart;
```

Listing 24: Basic validation of configuration in BD.

Other options are directly copied from the configuration object to the browser instance, as seen in listing 25. This introduces a risk of a user overwriting vital browser state.

The configuration and initialization processes of many parts of BD, both user-facing and internal, are woven into one single process. These processes are difficult to

```
for (var k in opts) {
    this[k] = opts[k];
}
```

Listing 25: Other parts of the configuration are not validated.

understand, as they conflate many different parts of program behavior, and have far-reaching consequences by passing options to other parts of the program without validation. There is also not a centralized specification of what options are valid or even what can be configured, as all parts of the provided configuration can be used by other parts of the program, as shown in listing 25.

These are problems GGB must avoid; the next section shows how. The configuration provided by the user is validated at the start of the program, providing errors that make it clear what went wrong, making it impossible to use an incomplete or incorrect configuration. The result is a configuration object whose type is defined in a single place; in this way there is a clear and canonical specification of the possible configuration options, even when other parts of the program actually perform the parsing and use the options.

4.3.2 A type for browser options

In listing 26 the type of the configuration object used to initialize GGB is defined, i.e., the type of value that the user-provided configuration is parsed to.

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

Listing 26: The ‘BrowserConfig‘ type defines the configuration options.

The exact contents of the `BrowserConfig` type are not important, what matters is that they are all PS types, and so can be used safely. Creating a value of this type is done by parsing a user-provided configuration, using the `parseBrowserConfig` function.

The type signature, shown in in listing 27, states that the function takes an unknown (foreign) JS value, and outputs either a `BrowserConfig`, or a `NonEmptyList` of

ForeignErrors[2]. NonEmptyList is the type of lists that have at least one element — the compiler ensures that the list cannot be empty. ForeignError is defined by the package purescript-foreign[3], which is a library that provides types and functions for working with foreign data (JS objects), including parsing them to well-typed PS values. Listing 28 shows the definition of ForeignError, which simply encodes some of the things that can go wrong when parsing an unknown JS value.

```
parseBrowserConfig
  :: Foreign
  -> Except (NonEmptyList ForeignError)
            BrowserConfig
```

Listing 27: Type signature of function that validates a user-provided configuration object.

In other words, the type of parseBrowserConfig says that it attempts to parse an unknown value into a browser configuration, and that if it fails to parse the provided value, it must provide at least one error message — silent failure is not an option. Implicitly, the type also states that each of the values in the browser configuration used by the main GGB instance must be derived and assigned in this function. It is the single source of truth, which its BD counterpart lacks.

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

Listing 28: The types used to encode errors when parsing.

Listing 29 shows part of the actual parsing machinery, namely, the part that parses and validates (on a very simple level) the BD browser constructor. In English, the name browser, which is later returned as part of the parseBrowserConfig output, is bound to the result of attempting to read the field with key name "browser" from the JS object provided: f is the JS object, ! is an indexing operator from purescript-foreign, which fails with an ErrorAtProperty if the field does not exist, communicating as much to the function caller.

If the field does exist, the next two lines ensure that it is a function. If it is not, a ForeignError is returned, with an error message that the "browser" key should have been a function.

[2]In ‘purescript-foreign‘, the type alias ‘F a = NonEmptyList ForeignError a‘ is used. The full type is used here for clarity.

[3]Available on Pursuit at
https://pursuit.purescript.org/packages/purescript-foreign

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

Listing 29: Basic validation on the provided BD constructor.

```
tracks <- f ! "tracks" >>= readTracksMap
bdRenderers <- f ! "renderers" >>= parseRenderers
pure $ BrowserConfig
  { wrapRenderer, bdRenderers, browser, tracks, events }
```

Listing 30: Basic validation on the provided BD constructor.

Listing 30 shows how two other fields are parsed; it is done analogously to the `browser` field. These fields are somewhat more complex, and so call out to other functions to finish the parsing. Finally, the `BrowserConfig` is returned by the function, a record wrapped in the newtype constructor defined in listing 26.

The parsing is done by sequencing the results of trying to parse each of the parts, and combining them in the record. If any of the parsers fail, the `parseBrowserConfig` function returns with the corresponding failure message. This is done by virtue of `Except` being an instance of the `Monad` typeclass; the `do`-notation, including the `<-` operator, are syntactic sugar for the functions provided by the `Monad` class, allowing us to combine effectful (in this case, potentially throwing `ForeignErrors`) computations[4].

4.3.3 Configuring Browser Data

Configuring a BD track versus a Cy.js graph are quite different tasks. They are both provided as arrays of JSON data, but obviously have different requirements, and are parsed and validated in different ways.

While Cy.js supports highly complex data, graphs in GGB are currently configured simply by providing a name and a URL from which to fetch the elements in JSON format.

Tracks using BD, intuitively, are configured with BD configurations; it is possible to copy the JSON that defines a track from a page using BD, to the GGB configuration, without modification.

Because BD supports so many different types of track, data formats, etc., GGB takes a hands-off approach to BD track configurations; the only validation that

[4]In general, it is preferable to use Applicative parsing instead of monadic, as it can attempt to parse the entire structure in parallel, and return *all* errors, not just the first. For an excellent introduction to this, see `https://github.com/jkachmar/purescript-validationtown`.

takes place is that a track must have a name. It is simply not feasible to perform more validation, due to the complexity of the relevant BD code.

4.3.4 Summary

One of the greatest problems with the configuration system in BD is that it provides very little information as to what options do what, or even what options are available, much less what values are legal for what options. With an incorrect configuration, things can go wrong in parts far from the parts of the code that manage configuration and initialization. These ring many bells concerning the legacy code problem of not understanding what code does, and difficulties of predicting the consequences of code.

The configuration system provided by GGB, on the other hand, collects all options in one place, and one type. Since it is type-checked, no part of the program can receive an invalid value. Validation is done in such a way that errors are discovered and reported before the program can attempt to use any of them, preventing silent failure, or failure in some part of the program far away from the configuration system.

The next section continues on the theme of increasing program correctness while staying compatible with BD, by introducing one way of differentiating values of different units.

4.4 Types and units

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. This section examines why that is the case, and how lifting units to the type level gives us additional program correctness.

4.4.1 ‘Number‘ iss not meaningful

BD uses the JS `Number` type, a double-precision IEEE 754 floating point, for all its numerical values. When all values are regular JS numbers, there is nothing to stop the programmer from e.g. adding a length to a weight, which is likely to lead to unexpected program behavior. It also provides no additional information on the meaning of what the function does in the program. For other types of data, BD uses mainly JS objects and strings.

One way to solve this problem and provide more correctness and information in JS would be to use something like the `daggy` library[5], 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 on a chromosome.

Since PS has a type system with built-in sum types, we expect it to be easier to represent these kinds of units. In fact, PS makes it possible to create new types that reuse another type's underlying runtime representation, which is highly efficient. This is done using newtypes, introduced in the next section.

4.4.2 Newtypes

Newtypes are one of the ways of creating types in PS, their name stemming from the `newtype` keyword used to create them. A newtype 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, meaning they can safely be passed to FFI functions, nor is there any performance hit in general. Examples of using newtypes to represent units in GGB come next.

4.4.3 Positions on a genome

BD uses basepairs (Bp) for all position data, though represented as a JS `Number` rather than an integer. It is 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! Hence GGB has types for these units, defined in listing 31.

For transforming between newtypes and their underlying wrapped type, PS provides

[5]‘daggy‘ can be found on GitHub at `https://github.com/fantasyland/daggy`

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

Listing 31: Bp and MBp type definitions.

the typeclass `Newtype` typeclass, which defines the functions `wrap` and `unwrap` to, in our example, move from a `Number` to a `Bp` or `MBp`, and vice versa. The compiler derives the instances for us, and in listing 32 an example of transforming from `MBp` to `Bp` is given.

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

bpToMBp :: Bp -> MBp
bpToMBp x = wrap $ unwrap x / 1000000.0
```

Listing 32: Transforming between units using the Newtype class. The compiler infers the ‘unwrap‘ function to have the type ‘MBp -> Number‘, and the ‘wrap‘ function to have the type ‘Number -> Bp‘.

PS also provides facilities for deriving typeclass instances for newtypes, using the instances on the wrapped type. For ease of use, listing 33 shows using this to derive equality, ordering, and arithmetic typeclasses. With these we can use regular arithmetic operations for adding, subtracting, etc. with these units[6].

```
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
```

Listing 33: Deriving instances

Now it is easy to manipulate values of these types. Listing 34 shows adding two basepair positions in the REPL, working as expected.

A basepair is a position on a chromosome, and a genome consists of several chromosomes. BD represents chromosomes by their string identifiers. As with `Bp`, we use a newtype, as seen in listing 35.

A genome browser must be able to convert from genome positions to screen locations. A scaling factor is one of the pieces of that puzzle, as detailed next.

[6]All of these are derived only for convenience, despite many not making much semantic sense considering the units. E.g. what does it mean to multiply two positions on a genome? (Not much.)

```
> p1 = Bp 123.0
> p2 = Bp 400.0
> p1 + p2 == Bp 523.0
true
```

Listing 34: Adding two 'Bp' values in the PS REPL.

```
newtype Chr = Chr String
derive instance newtypeChr :: Newtype Chr _
```

Listing 35: Representing chromosome identifiers as wrapped strings.

4.4.4 Scale

When drawing data to the screen, we need to be able to transform between screen coordinates and the coordinates used by data. As `Bp` and `MBp` are isomorphic[7], here we only look at transforming between basepairs and pixels. In listing 36 we represent this scaling factor with another newtype.

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

Listing 36: Definition of Bp/Pixel scaling factor.

Functions for using this type to transform basepairs and pixels are defined in listing 37. The next section uses these coordinate types to create a representation of a data feature, as used by BD.

4.4.5 Features

`Feature` is what BD calls basically any data point. While the feature objects in BD are quite complex, as various data parsers construct them in different ways, there are only four necessary pieces to them: what chromosome the feature is on, the start and end basepairs of the feature, and whatever data the feature contains, which may be arbitrary.

A data type consisting of a single possible value constructor containing multiple pieces of data is known as a "product type", as it is is isomorphic to the Cartesian product of the component types. Listing 38 shows the definition of the `Feature` type in GGB. The type takes two type parameters, `c` and `r`, corresponding to the coordinate and contained data, respectively. The `BDFeature` type alias concretizes

[7]This is not a true isomorphism, due to the various oddities and problems that are inevitable when dealing with IEEE 754 floating point arithmetic, but close enough for data visualization purposes.

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

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

Listing 37: Using the scaling factor to correctly transform between basepairs and pixels.

the type to use basepairs as coordinates.

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

type BDFeature r = Feature Chr Bp Bp r
```

Listing 38: The types of features in GGB and BD.

For convenience, we have the compiler derive how to compare two `Features` for equality and order. This is shown in listing 39. The equality and ordering defined on the coordinate and data that the feature consists of are used to achieve this. Additionally, the compiler can derive a Functor instance, with which we can easily transform the data in the feature if so desired.

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

Listing 39: Deriving instances on the 'Feature' type.

At this point, we have some types that let us work with data closely related to the BD representation, using units that both give safety and are easy to provide to the type-agnostic BD browser. There is no reason to stop here, however. We can further exploit the structure of the `Feature` type definition to gain some additional features.

As an example, the `Bifunctor` typeclass, provided by the `purescript-bifunctor` package[8], provides methods for applying functions to both parts of a compound data structure that has a `Bifunctor` instance. Our `Feature` type is a prime example, with the instance given in listing 41.

Now it is easy to modify either or both parts of a `Feature`, leaving the chromosome identifier intact.

Let us end this section with an example: listing 42 shows the entire definition of a function that transforms features with coordinates as basepairs to features with

[8]Available on Pursuit at
`https://pursuit.purescript.org/packages/purescript-bifunctors`

```
class Bifunctor f where
  bimap :: forall a b c d.
           (a -> b) -> (c -> d) -> f a c -> f b d

- | Map a function over the first type argument of a `Bifunctor`.
lmap :: forall f a b c.
        Bifunctor f => (a -> b) -> f a c -> f b c
lmap f = bimap f id

- | Map a function over the second type arguments of a `Bifunctor`.
rmap :: forall f a b c.
        Bifunctor f => (b -> c) -> f a b -> f a c
rmap = bimap id
```

Listing 40: Definition of ‘Bifunctor‘ typeclass and related functions, from `https://github.com/purescript/purescript-bifunctors/blob/v3.0.0/src/Data/Bifunctor.purs`

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

Listing 41: ‘Bifunctor‘ instance on our ‘Feature‘.

```
featureBpToMBp :: forall r.
                  Feature Bp r
               -> Feature MBp r
featureBpToMBp = lmap bpToMBp
```

Listing 42: Transforming feature coordinates.

coordinates as megabasepairs.

4.4.6 Summary

Various units concerning the position of data, all effortlessly compatible with BD while providing type-safety, have been presented, together with tools for transforming them.

A more complex data type representing the most basic building block of data points as BD sees them, the `Feature` type, was also trivially defined using the tools provided by PS, together with highly general functions for lifting transformations on the components of a feature to the whole. This was in fact a sneak peek at what the next section provides, which concerns transforming data to representations suitable to be displayed on-screen, and more.

4.5 Rendering and positioning data on the screen

This section concerns transforming data, e.g. values of the `Feature` type defined in the previous section, to representations that can be rendered to the screen, SVG, etc. In a word, transforming it to something useful to the user. First BD's solution is presented, and why it is difficult to work with.

The solution used by GGB is then presented, beginning with examining what the core of the problem is. A system to work with these transformations in the abstract is created, finally this system is then to perform all of the desired transformations in a modular and extensible manner.

4.5.1 Biodalliance

In BD, a "glyph" is something that can be drawn to the browser screen, 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. BD has a number of Glyphs, each of which is a class providing a basic interface:

- a method to draw the glyph to a canvas
- a method to produce an SVG element of the glyph
- methods for determining the bounding box of the glyph

The glyphs range from basic geometric shapes such as boxes and triangles, to more complex ones that work with other glyphs, e.g. a glyph that translates one or more glyphs. As glyphs are classes, each has their own constructor which takes the arguments required to define the glyph, such as position on screen, line and fill color, etc.

For clarity, we now take a closer look at the `BoxGlyph`, whose shape is obvious. Listing 43 shows its constructor, which simply uses the parameters to set the corresponding fields on the instantiated glyph.

```
function BoxGlyph(x, y, width, height,
                  fill, stroke, alpha, radius) {
    this.x = x;
    this.y = y;
 /* ... elided assigning each of the other parameters
        to fields on the BoxGlyph object (`this`)  */
}
```

Listing 43: The creation of a box glyph.

These fields are then used in the other methods. Listing 44 shows a slimmed down version of the method that draws a `BoxGlyph` to a provided HTML5 canvas. The data in the glyph is used when performing a set of canvas drawing instructions.

```
BoxGlyph.prototype.draw = function(g) {
    g.strokeStyle = this.stroke;
    g.lineWidth = 0.5;
    g.strokeRect(this.x, this.y, this._width, this._height);
}
```

Listing 44: Basic method to draw a glyph to a canvas.

Another method supported by glyphs is exporting to SVG, the `toSVG` method. Listing 45 shows the `BoxGlyph.toSVG` method. As expected, the glyph data is used to create an appropriate SVG element.

```
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'});
    return s;
}
```

Listing 45: Method to create an SVG element from a box.

The methods for determining the bounding box of a glyph are less obvious, as shown in listing 46. These are used for determining whether the user has clicked on a glyph. The methods are all constant functions, which lets different glyphs calculate their bounding boxes in different ways. However they are limited to axis-aligned bounding boxes.

```
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;
}
```

Listing 46: Bounding box-related methods on the box glyph.

A problem with this way of creating and working with glyphs is that it necessarily requires large amounts of duplicated code. Basic canvas instructions, SVG elements, and bounding boxes must be written for every glyph, of which there are dozens in BD. There is also simple way of combining existing glyphs; the best one can do is pass around multiple glyphs and call their respective functions one after another, and writing functions to combine bounding boxes. Conversely, adding a new glyph transformation requires modifying every single existing glyph, separately.

This is a problem to be avoided in GGB. A single interpretation-agnostic way of creating glyphs is created, in such a way that glyphs can be composed to create new glyphs — in other words, a simple domain specific language for glyphs. Interpreters for this language are then written to draw these glyphs to the canvas, SVG elements, and more. Best of all, pure FP and PS provide the tools to make this quite simple.

4.5.2 Glyphs in the Graph Genetics Browser

First, the type representing glyphs is presented, followed by a brief introduction to the "Free monad" abstraction, which our language is defined with.

We require some types to represent our glyphs. Listing 47 show these, a simple `Point` type representing a point in 2D space, and the `GlyphF` type which contains the commands in our DSL for constructing glyphs. We also derive a `Functor` instance for `GlyphF`. This is important, because our DSL is in fact the Free monad of this `GlyphF` functor.

```
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

derive instance functorGlyph :: Functor GlyphF
```

Listing 47: Type definitions for our glyphs.

The Free monad is named so because it is the monad that arises from any functor. A naive implementation (which works in Haskell thanks to non-strict evaluation, but not PS) is given in listing 48. Intuitively, especially when using it to create DSLs, it can be thought of as a list of commands to perform, where the commands are defined by the underlying functor. This list of commands can be used by interpreting it into some other functor.

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

Listing 48: Naive implementation of the Free monad.

First, however, we need to finish the DSL, so we have something to interpret. First we wrap our `GlyphF` functor in `Free`, with a type synonym to make things cleaner, in listing 49.

```
type Glyph = Free GlyphF
```

Listing 49: The Free monad on GlyphF.

Next we "lift" the value constructors of `GlyphF` into the Free monad DSL. A subset of the commands are given in 50, and the rest are exactly analogous.

```
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
```

Listing 50: Some of the lifted functions in our DSL.

Now we have a number of functions which produce values in our DSL, and can easily create and combine glyphs with it. As an example, in code block 51 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}
```

Listing 51: A simple glyph in our DSL.

Note that this glyph is entirely abstract; it is a syntax tree representing the action of constructing the glyph. The interesting part lies in interpreting this data structure, in 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.

It is time to look at such an interpreter. We begin with a simple one that transforms a `Glyph` into a string.

4.5.3 Logging Glyphs

Before writing an interpreter, let us look at how to run one. We do so with `foldFree`, the type signature for which is in 52. It takes a natural transformation from our DSL functor to the target functor. That is, an interpreter cannot touch the contents of the functor; it cannot look beyond the current instruction in the DSL.

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

Listing 52: Type signature for function that runs interpreters.

If we want to produce a string, we need to find a monad that has the effect of doing so. The Writer monad is a natural fit, and conveniently also has a MonadRec instance, and so can be used with `foldFree`. The type of the glyph-to-string interpreter is given in listing 53.

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

Listing 53: This interpreter performs its actions by producing strings, appended by Writer.

Listing 54 shows a subset of the function body. For each glyph primitive, it writes an appropriate string, and return the contents of the functor, which is the next "step" in our glyph "program".

```
glyphLogN (Stroke c a)    =
  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
```

Listing 54: Interpreting glyphs into strings.

Running the interpreter consists of applying this natural transformation using foldFree, then getting the resulting String from the Writer. The function `showGlyph` in listing 55 nearly writes itself at this point.

As an 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 listing 56.

All that remains now is writing more interpreters. First, the graphical ones, for canvas and SVG display.

4.5.4 Drawing Glyphs

When drawing to canvas, we use Eff as the target for our natural transformation. Interpretation is done by performing the appropriate canvas effects, see listing 57.

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

Listing 55: Function transforming arbitrary glyphs to strings.

```
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)
```

Listing 56: Output of logging an example glyph.

glyphEffN is then used in renderGlyph, in listing 58, to interpret an entire Glyph structure into a canvas instruction.

```
glyphEffN :: Context2D
          -> GlyphF
          ~> 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
- ..
```

Listing 57: Subset of the canvas interpreter.

SVG on the other hand interprets Glyphs into the SVG type, a monad transformer stack defined in listing 59.

The result is a series of commands which can be used to produce the desired SVG element. The interpreter is in listing 60, and is very similar to the HTML canvas interpreter in listing 57.

The interpreter is used in listing 61, first to map Glyphs to pure SVG elements, then to render the SVG elements using the DOM.

Only one part of the puzzle remains, namely producing bounding boxes for glyphs.

```
renderGlyph :: Context2D
            -> Glyph
            ~> Eff
renderGlyph = foldFree << glyphEffN
```

Listing 58: Function for drawing arbitrary glyphs to an HTML canvas.

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

Listing 59: SVGs are constructed by appending elements. Creating an element depends on the state of the SVG context, which contains information such as the current stroke color, transform matrix, etc.

```
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
- ..
```

Listing 60: Glyph to SVG interpreter.

```
runSVGEff :: forall a.
             Glyph a
          -> Array SVGElement
runSVGEff =
  execWriter << flip runStateT SVG.initialSVG
             << foldFree interpSVGEff

renderGlyph :: forall a.
               Glyph a
            -> Eff Element
renderGlyph = SVG.renderSVG << runSVGEff
```

Listing 61: Functions for creating SVG elements from glyphs.

4.5.5 Glyph 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 PS, the "bounding box" type would look like the type `BoundingBox` in listing 62. 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 }
```

Listing 62: The BD bounding box type in PS

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, and the binary operation denoted with $\diamond$ (written as `<>` in PS, called "append"). If this following law is true, we have a semigroup:

$$\text{Associativity:}\ (x \diamond y) \diamond z \ = \ x \diamond (y \diamond z)$$

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:

$$\text{Left identity:}\ x \diamond e \ = \ x$$
$$\text{Right identity:}\ e \diamond x \ = \ x$$

Examples of monoids again, arrays, lists, and strings, where the identity element is the empty array, list, or string. The natural numbers with addition form a monoid only if zero is counted among the naturals; without zero, it is only a semigroup. Another counterexample of a semigroup that is not a monoid is the non-empty list.

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 listing 63; a record keeping track of the four edges of the box.

```
newtype GlyphPosition =
  GlyphPosition { min :: Number
                , max :: Number
                , minY :: Number
                , maxY :: Number
                }
```

Listing 63: Newtype wrapper for bounding boxes.

`GlyphPosition` 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. The semigroup instance is shown in listing 64.

```
instance semigroupGlyphPosition
  :: Semigroup GlyphPosition where
    append (GlyphPosition p1)
           (GlyphPosition p2) =
      GlyphPosition
        { min:  Math.min p1.min  p2.min
        , max:  Math.max p1.max  p2.max
        , minY: Math.min p1.minY p2.minY
        , maxY: Math.max p1.maxY p2.maxY
        }
```

Listing 64: How to append bounding boxes.

Note the use of the the `min` and `max` value functions from the Math module, and that all the heavy lifting is done by them. 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 use positive and negative infinity, which exist in the IEEE 754 standard. Using the JS `infinity`, the identities in us the identities in listing 65.

Now the identity `GlyphPosition` is easily defined by setting the minimum sides to positive infinity, and the maximum sides to negative infinity, as in listing 66.

Now, with our `Monoid` instance in hand, we can write another interpreter for Glyph, using Writer as our monad in the natural transformation, see listing 67.

```
- for any number x
> Math.min x  infinity == x
true
> Math.max x -infinity == x
true
```

Listing 65: Identities on min and max using infinity, in the PS REPL.

```
instance monoidGlyphPosition
  :: Monoid GlyphPosition where
    mempty =
      GlyphPosition { min:    infinity
                    , max:  (-infinity)
                    , minY:   infinity
                    , maxY: (-infinity)
                    }
```

Listing 66: The identity of bounding boxes.

```
glyphPosN :: GlyphF ~> Writer GlyphPosition
glyphPosN (Stroke _ a) = pure a
glyphPosN (Circle p r a) = do
  tell $ GlyphPosition { min: p.x - (r * 1.5)
                       , max: p.x + (r * 1.5)
                       , minY: p.y - (r * 1.5)
                       , maxY: p.y + (r * 1.5)
                       }
  pure a
- ..
```

Listing 67: The bounding box interpreter.

Finally, in listing 68 this interpreter is used exactly as the previous Writer-based interpreters were.

```
glyphToGlyphPosition :: forall a.
                          Glyph a
                       -> GlyphPosition
glyphToGlyphPosition =
  execWriter <<< foldFree glyphPosN
```

Listing 68: Function for extracting the bounding box from an arbitrary glyph.

Now bounding boxes come for free with all `Glyphs`, and we have the tools required to create glyphs compatible with BD. First, however, it would be good to ensure that it actually is a semigroup and monoid we have created, by testing it.

Testing our monoid

Semigroups and monoids have laws, and in PS there are tools for testing such laws. To do this, the package `purescript-jack` is used, which is a property-based testing framework, like QuickCheck.

First, the type signatures of some utility functions to generate and render GlyphPositions are provided in listing 69. They are then be provided to the functions provided by Jack to generate test values and validate the properties described next.

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

renderGlyphs :: ThreeGlyphs -> String
genGlyphPosition :: Gen GlyphPosition
genThreeGlyphs   :: Gen ThreeGlyphs
```

Listing 69: Utility functions for testing bounding boxes.

The law all semigroups should abide is associativity. In Jack, we describe a Property asserting that changing parentheses do not change equality, in listing 70.

In addition, monoids require that the identity element in fact be left and right identity. Listing 71 shows the definition of this property.

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

4.5.6 PureScript glyphs in Biodalliance

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

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

Listing 70: Property testing the semigroup laws, c.f. semigroup law equation above.

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

Listing 71: Property testing the monoid law.

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

To do this, we exploit the fact that PS 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 is given in listing 72.

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. This function is given in listing 73.

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

```
writeGlyph' :: forall a c r.
               Maybe (Feature c r)
            -> Glyph a
            -> Foreign
writeGlyph' f g =
  toForeign
    { "draw":    mkEffFn1
                   $ \ctx -> Canvas.renderGlyph ctx g
    , "toSVG":   mkEffFn1
                   $ \_ -> SVG.renderGlyph g
    , "min":     const p.min
    , "max":     const p.max
    , "minY":    const p.minY
    , "maxY":    const p.maxY
    , "feature": f'
    }
  where p = unwrap $ glyphToGlyphPosition g
        f' = toNullable $
               (\(Feature chr min max _)
                  -> {chr, min, max}) <$> f
```

Listing 72: Composing transformations to create BD-compatible data.

```
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
```

Listing 73: Helper function for creating glyphs.

4.5.7 Summary

The biggest problem with BD's representation of glyphs is code duplication and difficulty of composition. If one wants to create a new glyph, several functions must be written, all very similar. Likewise, a change to a glyph requires making the same change in many places. To a more extreme extent, a modification to the browser in general may require rewriting or adding another method to each of the glyphs. That is, the size of the required changes is proportional to the number of different glyphs.

The solution used by GGB avoids all of these problems. Creating new glyphs is simple and pleasant by using the Free monad DSL. Glyphs created in this manner are "first class," i.e. they can be used exactly like the primitives `Circle` etc., providing an easy interface to creating arbitrarily complex glyphs.

If a new way of using a glyph is desired, one need only write an interpreter for each of the glyph primitives, and the number of primitives is likely to remain constant. If a new primitive is to be added, the size of the required changes is proportional to the ways in which glyphs can be used, which is quite tractable.

There are more transformations required in GGB. Here we were concerned with transformations from data to tangible representations of data; in the next section, the problem of transforming events produced by different browsers is faced.

4.6 Transforming and handling events

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.

Briefly, 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
2. A track source, mapping the raw event to one used by GGB
3. A track sink, consuming GGB events into some callback that performs effects on…
4. … 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. That is, we want to be able to validate and make sure only compatible sources and sinks are hooked together; a sort of type safety, except at runtime, not compilation time.

The next two sections describe the events used by BD and Cy.js, what data they contain and what requirements they entail; what GGB must provide. Next, the reasoning behind the configuration language is presented, followed by implementation and usage details.

4.6.1 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, but for our purposes we will only be concerned with one, namely, the feature itself. This feature is a JS object, and can contain any information that BD parsed from the raw data, meaning two features can be different, and be arbitrarily complex, yet the user can add event handlers that work with them.

4.6.2 Cytoscape.js

Cy.js has a greater array of exposed interactions and event handlers than BD, but we will, again, keep it simple. We focus on regular clicks, and so are interested in the `cy.on("click")` function, which can be used in a similar way as BD's `addFeatureListener`.

Handlers attached with `cy.on()` receive numerous parameters, including the element that caused the event, which is what we will be using. Analogous to the "feature" provided to the BD handler, the "target" value provided to the Cy.js handler contains the entire element clicked on. As with BD, this can can contain arbitrary data.

Both BD and Cy.js, then, produce events with unstructured information of arbitrary complexity — unstructured 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.

4.6.3 Graph Genetics Browser

GGB must be able to make use of both BD and Cy.js events, in a way that is user-configurable. It must provide an interface that makes it possible to describe what events consist of, i.e. what data they contain and what they can be used for. This interface must be universal enough to work with both BD and Cy.js, and simple enough for users to be able to express some semantics (such as position on a genome, or expression of some phenotype) of their JSON-encoded data.

In other words, the user must be able to describe, for example, "this is what a position on the genome looks like", and "this is how to find the position in a BD event". Then, a Cy.js event handler that is configured to filter the Cy.js graph when it receives a genome position, should do its thing when BD produces an event containing position data.

What would this configuration interface look like? JSON is the configuration language used elsewhere by GGB, and so is the natural choice here. There is also the fact that the events from BD and Cy.js are JSON objects to keep in mind.

The question then becomes, what is a good way of describing a mapping between "raw" (i.e. untouched BD or Cy.js elements) events, and semantically-annotated events? The answer is that it looks like a subset of the "raw" event JSON tree, flipped on its head and shaken somewhat.

The annotated event is a collection of key/value pairs. The configuration for such an event is, simply, the event itself, with the values swapped for strings annotating their eventual types. The rule for creating such an event, given a raw event (e.g. from BD), consists of, for each value in the annotated event, a path describing where in the raw event said value can be found.

What the user needs to provide, then, is two JSON trees, whose values are all strings. The structures of the trees are the same as that of the raw event (specifically, a subset of the raw event) and that of the annotated event. By providing what are essentially example raw events and annotated events, the user tells GGB how to transform the former into the latter; the semantics of this part of the system and the configuration are, in a sense, one-to-one. The fact that JSON is the language in both cases, makes the implementation even simpler.

The difference between the configuration and the actual events are, in the "raw" case, that each leaf node contains the name of the corresponding "hole" that value will fill in the annotated event, and in the annotated case, that the leaf node contains a string with the "type"[9] of the value.

The following sections provide more concrete configuration examples, and detail the implementation of these JSON transformation and parsing machinery. The implementation consists of two types, `TrackSource` and `TrackSink`. The former transforms events from browsers to GGB events, the latter handles received events on browser tracks.

4.6.4 TrackSource

The definition in PS of a the `TrackSource` type can be seen in listing 74. It is a newtype 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 TrackSources 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
```

Listing 74: Definition of TrackSource.

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. BD).

For parsing all this JSON, the `purescript-argonaut` package[10] was used. The next section describes how the TrackSource event templates are parsed, and how JSON

[9]The word "type" is in quotes here, as there is no correspondence between the leaf node values and PS types, nor is there (as of yet) any kind of runtime "type"-checking implemented in GGB.

[10]Available on Pursuit at
`https://pursuit.purescript.org/packages/purescript-argonaut`

structures are used to work with other JSON structures.

Parsing templates and events

Argonaut is a library for working with JSON in PS, including serializing and de-serializing, as well as working with JSON trees. In this case we are interested in walking arbitrary JSON trees and transforming collections of paths.

Listing 75 shows an example of a `SourceConfig`, which describes how to parse an event such as the one in listing 76, to the object in listing 77.

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

Listing 75: Example SourceConfig, mapping a feature from BD to a range of basepairs on a chromosome.

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

Listing 76: A raw event from BD.

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

Listing 77: A parsed event constructed using the BD event.

These are simple (and real) examples, however the templates provided can be of arbitrary depth and complexity; as mentioned earlier, the only rule is that each leaf is a string, and that all field names are strings as well.

To determine how to create these annotated events, we extract a list of each of the key/value pairs in `eventTemplate`, and, for each one, we create a path to where the corresponding value will be placed in the finished event.

Next, to find out how to transform the raw event into an annotated one, the path to each leaf in the `rawTemplate` is extracted, and the named with the leaf node value. The result is a a mapping from keys in the annotated event, to a path describing where in the raw event the corresponding value can be found.

Argonaut provides functions for extracting and manipulating exactly JSON paths like this. The JCursor type, seen in listing 78, represents a path to a point in a JSON tree, at each step describing which key name or array index to go to next. As an example, listing 79 shows a JSON object and accessing a deep part of the tree, and the same path as a JCursor.

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

Listing 78: JCursor definition.

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

let cIs = thing.x[1].b.c; // (cIs == c) == true

// or as a JCursor:
JField "x" (JIndex 1 (JField "b" (JField "c" JCursorTop)))
```

Listing 79: JSON tree and accessor example.

Argonaut provides the `toPrims` function (type in listing 80) for transforming a JSON object into a list of pairs of paths to each value in the object (`JsonPrim` represents a JSON primitive value, i.e. anything that is not an array or map). Incidentally, this is exactly what we want to do with `rawTemplate`.

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

Listing 80: Creating cursors from a JSON tree.

The `eventTemplate` component is more complex, as it is not the leaf value itself that is desired, but the label of the path leading to it. In this case a step into the `JCursor` structure is required, as seen in listing 81.

`insideOut` is a Argonaut library function that reverses a `JCursor` — once more, a simple JSON library function is exactly what is required to solve our problem. We also ensure that the name is in fact a `String`, returning it wrapped in `Just`. If the leaf value was some other primitive, including `null`, `undefined`, etc., `Nothing` is returned.

```
insideOut :: JCursor -> JCursor

eventName :: JCursor -> Maybe String
eventName c = case insideOut cursor of
                JField s _ -> Just s
                _          -> Nothing
```

Listing 81: Grabbing the label of a leaf.

As this will be used in configuration, it is desirable to provide the user with information on what went wrong, and a value of `Nothing` does not say much. Thus, to provide the user with additional help when configuring, the source configurations are validated to make sure the given JSON structures "match", and errors are signaled using `Either String` instead of `Maybe`, providing some information.

Given any value that is going to be part of the annotated event, and all of the values we know we can get from the raw event, the name of the first value should be among names of the latter. If not, something is wrong, and some information can be provided to the user.

Listing 82 shows the implementation of the template validation function. The `Cursor`s are grabbed from the result of `toPrims`; the `JCursor`s themselves are unaltered.

```
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"
```

Listing 82: Validating templates.

To expand this to validate the array of cursors that define an annotated event, we use the fact that `Either` is an instance of the `Applicative` typeclass, and use `traverse`, as in listing 83.

In English: if our collection of templates `rcs` contains a rule explaining where in a raw event to find the desired value at path `vc` in the annotated event, return the `vc` path to the value; if not, throw an error.

The function tries to validate all given templates, and returns the first failure if there are any; we get validation of a collection of things practically for free. The

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

Listing 83: Easily expanding from one to multiple.

TrackSink concept, and its type, `TrackSink`, was implemented using similar tools, and is detailed next.

4.6.5 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 JS to PS are type-safe. BD and Cy.js TrackSinks, respectively, should have the types in listing 84.

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

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

Listing 84: TrackSink types.

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 listing 85 a BD TrackSink is defined that scrolls the BD viewport upon receiving an event.

These functions can be provided to GGB when configuring it. The next section describes how they actually are used to allow communication and interaction.

4.6.6 Using TrackSource and TrackSink

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 listing 86.

```
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 } ];
```

Listing 85: BD track sinks for moving viewport when receiving event with location data.

```
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 }
```

Listing 86: Function for creating a TrackSource from a SourceConfig.

To extend the above function to work on a collection of configuration objects, function composition is used in listing 87 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
```

Listing 87: Creating a TrackSource from multiple configurations.

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. Listing 88 shows the function that does so.

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

Listing 88: Function for parsing a raw event with a TrackSource.

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 `Event`s.

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, shown in listing 89.

Using a collection of `SinkConfigs` to produce a single TrackSink is not in itself complicated; see the code is in listing 90. The bulk of the logic is in validation, namely ensuring that there are not multiple handlers for a given event:

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

Listing 89: Creating a TrackSink from a sink configuration.

```
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
```

Listing 90: Validating and creating a TrackSink from multiple configurations.

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. The function is defined in listing 91.

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

Listing 91: Function to choose function to run when a TrackSink receives an event.

However, since TrackSinks are intended to perform effects, a helper function for that is useful. In particular, the function `forkTrackSink` in listing 92 asynchronously reads from a message bus, 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
```

Listing 92: Helper function for running functions when receiving asynchronous events on a bus.

4.6.7 Summary

TrackSource and TrackSink provide highly general ways of defining interactions between different parts of GGB. They are easily configured in a declarative manner by providing JSON templates, and there is a good degree of verification on their configuration.

In the next section, we finish our tour through the codebase of GGB by looking at how all these pieces fit together when creating a UI.

4.7 User interface

The construction of the UI of GGB is presented, beginning with a look at some of the problems with BD's approach. After, the UI library used in GGB is introduced, after which the components of the UI are presented and composed, including using TrackSink and TrackSource for communication between BD and Cy.js.

4.7.1 Biodalliance

BD has a full-featured UI for exploring genomic data chromosome-wise, adding and removing displayed tracks, configuring the browser, and exporting the current view to publishable 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 PS, 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.

4.7.2 The Halogen library

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), shown in listing 93.

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

Listing 93: The basic halogen component type.

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 do not need any, and 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 is 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 query algebra for the main component of GGB is given in listing 94.

```
data Query a
  = CreateBD (HTMLElement
               -> Eff Biodalliance) a
  | PropagateMessage Message a
  | BDScroll Bp a
  | BDJump Chr Bp Bp a
  | CreateCy String a
  | ResetCy a
```

Listing 94: The main browser query algebra.

From top to bottom, we can ask it to `CreateBD`, providing a function that creates a BD 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.

Listing 95 shows the type of the `eval` function in our main component. This is the function that maps queries to Halogen actions.

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

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

Listing 95: The type of the main component evaluation function.

child components. Listing 96 shows those in the main GGB component they are.

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

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

Listing 96: Main component child slot and query types.

Listing 97 shows the start of the `eval` function. 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. `H.action` is a Halogen function mapping ChildQuery constructors to concrete actions, by simply applying the `Unit` type to it. Finally, the next command is returned.

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

Listing 97: Evaluation of ‘CreateBD‘ query.

The next query is `PropagateMessage`, which receives a Message sent from the function handling messages from the children. The messages handled by the main GGB component can be seen in listing 98.

The evaluation of `PropagateMessage` is shown in listing 99. Depending on which message it is, we print a log message, and then use `H.raise` to send the message out from Halogen.

The rest are simple queries to the respective child component, practically the same as `CreateBD`. See listing 100.

Now we have a Halogen component that knows exactly what to do, however it cannot

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

Listing 98: Messages handled by the GGB main component.

```
PropagateMessage msg next ->
  H.raise msg *> pure next
```

Listing 99: Messages are handled by passing them along.

```
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
```

Listing 100: Most of the actions delegate to the BD and Cy.js components.

actually draw anything to the screen. This is dealt with next.

Rendering

Rendering the component 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, see listing 101.

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

Listing 101: The type of the main rendering function.

The function itself is simple, constructing the HTML tree with arrays of functions for creating HTML elements. Listing 102 shows a version simplified for brevity.

```
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]
    ]
```

Listing 102: Subset of the main rendering function.

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. The messages the BD and Cy.js components can produce are shown in listing 103.

```
data UIBD.Message
  = SendBD Biodalliance

data UICy.Output
  = SendCy Cytoscape
```

Listing 103: Messages produced by BD and Cy.js components.

Note that the main container uses its own messages (from listing 98) to propagate the children components; message passing is limited by Halogen, and anything more complex than this should be done on another channel. This is what is what GGB does with events. The messages from the BD and Cy.js components are handled by the functions `handleBDMessage` and `handleCyMessage` in listing 104.

```
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)
```

Listing 104: Propagation of messages in main component.

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 calls `H.raise` on the message, sending it to the outside world.

Creating the main component

These functions, including one to produce the initial state, 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.

4.7.3 The main application

Listing 105 shows the type and beginning of 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 effects in Eff. In our case, effects such as being a genetics browser.

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

Listing 105: Type of the main function.

In listing 106 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
```

Listing 106: Running the browser configuration parser.

With a validated config, we can create the track/graph configs, and create the function which will later be used to create Biodalliance, in listing 107.

```
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 :: (HTMLElement -> Eff Biodalliance)
    mkBd = initBD opts' config.wrapRenderer config.browser
```

Listing 107: Validating tracks and reporting eventual errors.

After picking the element to run in, we create the Halogen component, and create the Buses to be used by the events system. This is shown in listing 108. Note that we bind the value of `runUI` to `io`. `io` can be used to subscribe to messages sent from the main component, as well as send queries to it, which we do momentarily.

In listing 109 we use the provided TrackSink and TrackSource configurations to

```
io <- runUI component unit el'

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

Listing 108: Running the UI and creating event buses.

create the BD TrackSink and TrackSource, adding an error message if something went wrong.

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

Listing 109: Creating the BD TrackSink and TrackSource.

Finally, in listing 110, 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. After that is set up, the main Halogen component is told to initialize the BD browser.

```
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

    pure Nothing

  _ -> pure $ Just unit

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

Listing 110: Hooking of TrackSink and TrackSource and starting BD.

If the TrackSink was correctly configured, `forkTrackSink` is used to pipe events from the Cytoscape.js instance to the handler defined by said TrackSink. `forkTrackSink` returns a `Canceler` that can be used to kill the "thread", which is not needed, so we throw it away with `*> pure unit`.

Conversely, the TrackSource is used with the helper function `subscribeBDEvents`, defined in 111. It adds an event listener to the provided BD browser instance and writes the successful parses to the provided Bus.

```
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
```

Listing 111: Helper function to produce events from BD.

The code to set up the Cy.js TrackSource and TrackSink, and the Cy.js browser instance, is analogous, and elided.

4.7.4 Summary

The UI defined here wraps both BD and Cy.js, allows communication between them and GGB, and does this in a type-safe manner. The tools provided by Halogen makes it possible to construct complex UIs with a great degree of correctness, and the `Aff` monad makes it trivial to write asynchronous code.

4.8 Summary of results

Many problems related to legacy code were encountered during development, on various levels, from nitty-gritty line-by-line code to large modules such as configuration; others were potential problems, likely to rear their head when implementing software in domains like that of GGB. The problems 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 certainly not least, it is an extensible and flexible 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.

5 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.

5.1 PureScript and legacy JavaScript code

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

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

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 listing 72.

`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 achieving the development 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.

5.2 Functional and quality code

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. Listings 112 and 113 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 114 shows the implementation.

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

Listing 112: A concrete type for some function on an array of lists

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

Listing 113: A general type for some function on some foldable of a monoid

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

Listing 114: 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.

5.3 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.

5.4 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 had been 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.

6 Conclusion

Pure FP is 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.

6.1 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

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

studies are undertaken in the future.

More generally, there has been little research performed on 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.

6.2 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.

References

[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 users guide, 2005.

[24] A. M. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2 (4):153163, 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.