TODO thesis name

Christian Fischer

November 30, 2017

Contents

Introduction

> Legacy code. The phrase strikes disgust in the hearts of programmers. It conjures images of slogging through a murky swamp of tangled undergrowth with leaches beneath and stinging flies above. It conjures odors of murk, slime, stagnancy, and offal. Although our first joy of programming may have been intense, the misery of dealing with legacy code is often sufficient to extinguish that flame.
> — Robert C. Martin, Foreword to Working Effectively with Legacy Code [5]

1 Objective

2 Delimitations

3 "Background" [TODO: Rename]

4 GeneNetwork 2/Affiliations

Background/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. Next, an introduction to the purely functional programming paradigm

5 Legacy code

5.1 Definition

There is no formal definition of "legacy code", but [5] gives the definition "Legacy code is code that we've gotten from someone else." Bennett gives another definition, of legacy systems, which gives more of an idea of what the problem is: "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 in the trenches:

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

In other words, legacy code is code that continues to be relevant, e.g.\ by providing a service that is important, and that requires modification, or will require modification. If there were never going to be any reason to modify the code, it would not be worth talking about, nor is it likely that a system that provides a continually valuable service will not at some point in the future require maintenance or new features [6].

For this very 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 [8].

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" to be 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. [5] identifies four general reasons:

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

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

The main reason it is difficult to work with legacy code is lack of knowledge of the system and codebase, and how the system's behavior relates to the underlying code. Legacy codebases often lack documentation and tests, without which a new programmer on the project has few, if any, tools at their disposal to understand the codebase as it is, since they do not have any knowledge of how and why the code came to be as it is. Even if there is a design or system specification — which is far from certain — 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 [5] [1] [9]. 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 [10]. 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 — where the new system becomes another legacy system.

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.

Not only is working with legacy code difficult and expensive, it may be the case that doing so will not solve the underlying problems, dooming the legacy system to a lifetime of being subject to the swearing of programmers.

One question remain: is there anything special with legacy code, is *all* successful code doomed to one day bear the "legacy" label, or is it possible to write code that avoids, or at least reduces, the problems associated with legacy code? In other words, what characterizes the source code of legacy systems, and could code be written in another way?

Legacy code is difficult to work with; its dual would be code that is easy, even pleasant, to work with. Let us call that code "good", and the code that is characteristic of (but hardly unique to) the problems related to legacy code, "bad" code.

The question then becomes, what makes code good or bad? This is what the next section attempts to answer.

5.2 Code quality

In a word, we want heuristics for good vs. bad code. These are called code smells, or antipatterns.

First, however, we need to define what is meant by "good" or "bad" code.

Beck & Fowler provides a list of 22 code smells that have been used (extensively?) since publication.

Many of the code smells they list, as well as the solutions to them, are concerned about class-based object-oriented programming (OOP). OOP has been the primary programming paradigm for decades (TODO REF), WIP more here (but what?)

> 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. [10]

- Duplicated code - when a piece of code appears multiple times in the codebase, it's a sign of a potential abstraction. It also makes it difficult to change things, as the change needs to be duplicated several times – which also leads to more opportunities for mistakes to be made.

- primitive obsession - Using primitive types to represent values that could be better represented by composite types or wrapped types. This can lead to values being used where they shouldn't be (e.g. providing a Number representing a pixel to a function expecting a Number actually representing the number of objects in an array). This also makes it more difficult to understand what a variable or value is to the program.

- shotgun surgery - When functionality is spread out in the codebase in such a way that making a change at one place requires making a change at many other places. Reduces code comprehension, as distant pieces of code are somehow interacting, and makes it more difficult to modify the codebase successfully.

Simply, a good piece of code makes it clear what it does, how it relates to the system at large, and how it can be changed or reused without compromising its behavior or the behavior of the system.

5.3 Solutions

Reverse engineering of legacy code exposed `https://doi.org/10.1145/225014.225045`

"Reverse engineering of large legacy software systems generally cannot meet its objectives because it cannot be cost-effective. [..] it is very costly to “understand” legacy code sufficiently well to permit changes to be made safely, because reverse engineering of legacy code is intractable in the usual computational complexity sense."

As one of the main problems of legacy code is lack of knowledge, one of the main ways to attempt to solve it is to reverse engineer the existing system. This is usually done manually [TODO REF], but can also be done using automatic analysis of the code.

1. Code analysis One common way is to analyze the code to find ways to modularize it, to decouple the pieces from one another. This can be done by OOP stuff

   Another interesting route is finding a modularization 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.

2. Manual reverse engineering this section is not even a section.

   The most common solution is simply to do it by hand. This requires that programmers look at, and comprehend, the codebase, and how the codebase relates to the semantics of the system. Writing tests should also be done to give greater confidence when changes are made.

6 Purely functional programming

Functional programming as a paradigm focuses on functions in the mathematical sense, where functions, with some given input, always produces the same output. In purely functional programming, this concept taken to its limit, with functions not being able to perform side-effects, such as reading input from the user, or updating the user interface; more on these actions below. This is in contrast to the imperative paradigm, which places no such limitations on functions, other than scoping.

6.1 Functional programming

The functional paradigm can be seen as a natural extension to the lambda calculus, a model of computation invented by Alonzo Church, while imperative programming is closely connected to the von Neumann-style of computer on which it runs, and is similar to the idea of a Turing machine. Whereas a Turing machine models computation as manipulating symbols on an infinite tape of memory given a set of instructions, the lambda calculus models computation as function abstraction and application; the name derives from using λ to define functions.

The Turing machine and lambda calculus models of computation are (probably) equivalent, by the Church-Turing thesis. Thus any program that can be run on a theoretical Turing machine can be transformed to "run" in the lambda calculus. However, while programming languages that are built on the idea of a Turing machine are notoriously difficult to develop for, and are generally interesting only as curiosities or for research, languages based on lambda calculus are more wide-spread. Indeed, the pure functional language Haskell is at its theoretical core a typed lambda calculus, based on System-Fω, which it compiles to as an intermediate language [7].

6.2 Purity

Pure FP provides something called "referential transparency", which means that changing a piece of code to the result of running that code does not change the program. This makes "equational reasoning" possible, letting the programmer reason about parts of the program code as separate from the rest of the program. It gives the programmer confidence in what a function does.

Functional programming is orthogonal from type systems, but powerful type systems are closely related to pure functional languages. Haskell, being based on a typed lambda calculus, is a statically typed language,

and is an example of using a powerful type system to capture effects that are performed by the program – that is, letting a purely functional language express effects such as interacting with the real world

Besides capturing effects, a powerful type system provides the programmer with tools to increase productivity, decrease bugs, make refactoring easier, and improve the programming experience in multiple ways.

For example, Haskell and many(most?) other languages with similar type systems, do not have a 'null' value, instead encoding the possibility of lacking a value in the type system. In Haskell, the type 'Maybe' captures this possibility; if a function produces an 'Int', you can be sure that after calling the function you do indeed have an 'Int'.

A lower-level part of pure FP, which has seen increased use outside of FP(TODO REF) is immutability of data. In a purely functional language, functions cannot modify data passed to them, as doing so would be to perform a side-effect – passing the same variable to two functions would not necessarily have the result expected if one function can modify the input to the other. Using data structures that are immutable by default makes reasoning about programs much easier as it removes that possible side-effect, no matter the programming paradigm.

6.3 Advantages

As a type system gains features, the number of abstractions that can be expressed in it increases. Category Theory is a highly abstract branch of mathematics concerned with 'categories' consisting of 'objects' and 'morphisms' between objects. It is a rich field of research, and has over 70 years of results – and ideas and abstractions from it has been used in programming, especially pure FP. A classic example is Haskell's use of 'monads', an abstraction which captures the essence of sequential computation. Haskell uses a monadic type for its IO system.

If a programmer can express their problem in the language of category theory, they gain access to 70 years of documentation concerning their problem. If the abstractions used can be expressed in the type system, the compiler can help prove that the program is correct.

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 [REF Why functional programming matters]. You also get some level of program correctness, by writing code using types that correspond to the program semantics. You're able to construct transformations between data structures and compose them together – all type-checked by the compiler.

7 Functional programming for legacy codebases

Method

8 Biodalliance

9 Purescript

10 Genetics Graph Browser/Evaluation

Results

11 Interfacing with existing JS

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

11.1 FFI intro

Purescript'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. Here's an example of an FFI function which takes two values and returns their (JS-y) concatenation. In Purescript we normally have to make sure it makes sense to transform a value to a String before we print it, but Javascript has no such qualms:

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

Since JS doesn't care about the types, neither do we. The type signature is polymorphic in its two arguments, and returns an effect:

```
foreign import showStuff :: forall a b. a -> b -> Eff _ Unit
```

We can also define types (and kinds, and things of other kinds) using the 'foreign import' syntax:

```
foreign import data JSType :: Type
```

Now, the type ‘JSType‘ doesn’t have any data constructors in Purescript, so we can only create values of this type by writing an FFI function that returns it. Nor can we inspect the type without the FFI; to PS, it is entirely opaque.

11.2 Biodalliance

To work with Biodalliance, we define a foreign type corresponding to instances of the BD browser:

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

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

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

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

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

```
newtype BDEvent = BDEvent Json

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

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

11.3 Cytoscape.js

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

```
foreign import data Cytoscape :: Type

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

newtype CyEvent = CyEvent Json

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

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

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

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

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

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

```
-- | Add a Collection of elements to the graph
foreign import graphAddCollectionImpl :: forall eff.
                                         EffFn2 (cy :: CY | eff)
                                         Cytoscape
                                         (CyCollection Element)
                                         Unit

graphAddCollection :: forall eff.
                      Cytoscape
                   -> CyCollection Element
                   -> Eff (cy :: CY | eff) Unit
graphAddCollection = runEffFn2 graphAddCollectionImpl

-- | Get all elements in the graph
foreign import graphGetCollectionImpl :: forall eff.
                                         EffFn1 (cy :: CY | eff)
                                         Cytoscape
                                         (CyCollection Element)
```

```
graphGetCollection :: forall eff.
                      Cytoscape
                   -> Eff (cy :: CY | eff) (CyCollection Element)
graphGetCollection = runEffFn1 graphGetCollectionImpl


foreign import graphRemoveCollectionImpl :: forall eff.
                                            EffFn1 (cy :: CY | eff)
                                            (CyCollection Element)
                                            (CyCollection Element)

graphRemoveCollection :: forall eff.
                         CyCollection Element
                      -> Eff ( cy :: CY | eff) (CyCollection Element)
graphRemoveCollection = runEffFn1 graphRemoveCollectionImpl
```

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

```
-- | Apply a layout to the graph
foreign import runLayoutImpl :: forall eff.
                                EffFn2 (cy :: CY | eff)
                                Cytoscape
                                Layout
                                Unit

runLayout :: forall eff.
             Cytoscape
          -> Layout
          -> Eff (cy :: CY | eff) Unit
runLayout = runEffFn2 runLayoutImpl
```

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

```
newtype Layout = Layout String

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

1. Events

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

   ```
   newtype CyEvent = CyEvent Json
   ```

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

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

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

2. CyCollection

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(a) Tests

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

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

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

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

12 Configuration

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

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

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

12.1 Configuring Biodalliance

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

A very basic browser configuration could look like this:

```
var biodalliance = new Browser({

      prefix: '../',
      fullScreen: true

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

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

});
```

This object contains configuration of basic browser functionality (the properties ‘prefix‘, which is the relative URL for icons and such data, and ‘fullScreen‘ which controls how the browser itself is rendered); initial browser state (‘chr‘, ‘viewStart‘, ‘viewEnd‘, which together define the chromosome and range of 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 a mouse genome sequence is the only track.

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

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

1. Code Examples

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

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

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

this.viewStart = opts.viewStart;
```

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

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

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

    (b) Entangled types (probably remove this)

    Sources and styles are both tightly coupled and separate – problematic…

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

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

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

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

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

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

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

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

```
if (thisB.isDragging && rx != dragOrigin && tier.sequenceSource) {
    var a = thisB.viewStart + (rx/thisB.scale);
    var b = thisB.viewStart + (dragOrigin/thisB.scale);

    var min, max;
    if (a < b) {
        min = a|0; max = b|0;
    } else {
        min = b|0; max = a|0;
    }

    thisB.notifyRegionSelect(thisB.chr, min, max);
}
if (hit && hit.length > 0 && !thisB.isDragging) {
    if (doubleClickTimeout) {
        clearTimeout(doubleClickTimeout);
        doubleClickTimeout = null;
        thisB.featureDoubleClick(hit, rx, ry);
    } else {
        doubleClickTimeout = setTimeout(function() {
            doubleClickTimeout = null;
            thisB.notifyFeature(ev, hit[hit.length-1], hit, tier);
        }, 500);
    }
}
```

All of that to handle double clicks. Using purescript-behaviors, we could define an Event on double clicks by composition (I think). Compare to debouncing a switch with electronics vs assembly (maybe).

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

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

2. Another approach

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

12.2 BrowserConfig

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

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

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

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

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

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

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

```
type F = Except (NonEmptyList ForeignError)

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

‘Except‘ is practically ‘Either‘, and lets us represent and handle exceptions within the type system. In this case, the error type is a non-empty list of these possible error values. If something has gone wrong, there is at least one error message connected to it; it is simply impossible to fail a parse without providing an error message!

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

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

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

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

‘readTaggedWithError‘ is actually simple:

```
-- The type is:
readTaggedWithError :: forall a. String -> String -> Foreign -> F a
-- The implementation:
readTaggedWithError s e f = withExcept (append (pure $ ForeignError e)) $ unsafeReadTagged
```

In words, it tries to read the tag, and if unsuccessful, appends the provided error message to the error message from unsafeReadTagged. Let's look at the types:

```
unsafeReadTagged :: forall a. String -> Foreign -> F a

withExcept :: forall e1 e2 a.
               (e1 -> e2)
            -> Except e1 a
            -> Except e2 a

append :: forall m. Monoid m => m -> m -> m
```

In this case (of the type F), the use of 'withExcept' would specialize to have the type:

```
withExcept :: a.
              (a -> a)
            -> F a -> F a
```

Another way to look at it is that 'withExcept' is 'map' but for the error type.

12.3 Tracks

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

1. Biodalliance

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

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

   The Biodalliance constructor is another parameter that the configuration requires. This and the 'wrapRenderer :: RenderWrapper' function are required for the BD interface to function properly, and are JS functions provided by Biodalliance. (TODO note that wrapRenderer is only in a modified repo?)

2. Cytoscape.js

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

12.4 Events

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

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

1. Parsing the user-provided SourceConfigs

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

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

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

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

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

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

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

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

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

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

Putting it all together, we have this natural transformation:

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

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

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

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

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

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

(TODO: should probably just validate in the parseSourceConfig) Note how we discard (_ <- ...) the results from the config validation; we only care about the validation error, since the configuration values have already been parsed.

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

13 Units

13.1 Biodalliance/The problem

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

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

WIP this is the case in Biodalliance BD uses mainly raw JS numbers and strings for representing its state and data, with a few JS objects used mainly for more complex information.

TODO examples

WIP "solutions" in JS – tagged objects One way to solve this problem in JS would be to use something like the ‘daggy‘ library [TODO footnote: `https://github.com/fantasyland/daggy`], which adds tagged sum "types" to JS. The developer still needs to make sure they are used correctly, but at least the program will fail with an error if a value representing a pixel length is supplied to a function expecting a length in basepairs.

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

13.2 Newtypes

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

13.3 Positions

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

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

TODO add link to lines in Units.purs

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

To work with these, we can use pattern matching:

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

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

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

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

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

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

TODO: maybe note that most of this doesn't make very much sense, e.g. multiplying two Bp's is in fact pretty silly. Would probably be "better" to use a semigroup where <> is addition...

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

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

p1 + p2 == Bp 523.0
```

TODO: needs a super basic lens primer somewhere (maybe just a footnote in the first use of it), plus readBp might not be correct The Newtype instance also gives us access to the _Newtype lens isomorphism:

```
_Bp :: Iso' Bp Number
_Bp = _Newtype

_MBp :: Iso' Bp Number
_MBp = _Newtype

readBp :: String -> Maybe Bp
readBp s = s ^? _Number <<< re _Bp
```

13.4 Chromosomes

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

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

13.5 Scale

NOTE: This is currently only used in the Native track, however the old BD rendering stuff could/should be refactored to use the new BpPerPixel

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

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

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

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

13.6 Features

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

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

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

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

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

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

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

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

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

14 Glyphs

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

14.1 Biodalliance

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

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

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

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

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

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

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

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

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

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

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

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

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

    return s;
}
```

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

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

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

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

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

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

14.2 Glyphs in the Genetics Graph Browser

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

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

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

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

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

The Free monad is named so because it is the

In Haskell, the definition is very simple, thanks to non-strict evaluation:

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

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

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

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

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

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

```
type Glyph = Free GlyphF
```

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

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

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

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

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

-- and so on
```

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

Now we have a number of functions which produce values in the type ‘Free GlyphF‘. With them, we can use Purescript’s do-notation, and all the other tools that come with the Monad typeclass. Here we create a simple glyph that consists of three primitives:

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

A Glyph, then, is simply a data structure. The interesting part lies in interpreting this data structure; or, in other words, transforming it into another data structure, especially one that performs effects. In fact, an interpreter consists of a natural transformation from the ‘GlyphF‘ functor to some other functor.

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

14.3 Logging glyphs

The GlyphF.Log interpreter transforms Glyphs to Strings, which we can then log to the console. To run an interpreter, we use foldFree:

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

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

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

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

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

```
glyphLog :: GlyphF ~> Writer String
```

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

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

glyphLog (Circle p r a) = do
  tell $ "Drawing circle at (" <> show p.x <> ", " <> show p.y <>
         ") with radius " <> show r <> "."
  pure a
-- similar for the rest
```

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

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

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

For example, logging the process of drawing the previously defined ‘crossed-Out‘ glyph at the point ‘{ x: 40.0, y: 10.0 }‘ with radius ‘3.0‘ would produce the following output:

```
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)
```

14.4 Drawing glyphs to canvas and SVG

When drawing to canvas, we use Eff as the target for our natural transformation, and simply perform whatever canvas effects are appropriate:

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

```
                    , end: 2.0 * Math.pi
                    }
  _ <- C.stroke ctx
  _ <- C.fill ctx
  pure a
-- and so on

-- | Produce an effect to render the glyph to a canvas
renderGlyph :: forall eff. Context2D -> Glyph ~> Eff (canvas :: CANVAS | eff)
renderGlyph = foldFree <<< glyphEffN
```

SVG on the other hand uses the following type as target functor:

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

The result is a series of commands which can be used to produce the desired SVG element; this can then be rendered to the DOM:

```
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
-- and so on

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

-- | Render a glyph to an SVG element
renderGlyph :: forall a eff. Glyph a -> Eff ( dom :: DOM | eff ) Element
renderGlyph = SVG.renderSVG <<< runSVGEff
```

14.5 Generating bounding boxes

BD produces events when clicking on glyphs – which GGB make use of. To do this, BD expects four constant functions on each glyph. In Purescript, the "bounding box" type would look like this, and could be used directly by BD:

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

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

1. Semigroups and monoids TODO: <> can be rendered nice in latex, look that up Semigroups and monoids are concepts from abstract algebra and

category theory, however they are immensely useful in pure FP, as they appear in many different areas.

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

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

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

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

Now we can explore how bounding boxes form a monoid.

2. Monoidal bounding boxes TODO: ref to monoids and diagrams functional pearl

   The type corresponding to a glyph's position is GlyphPosition:

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

   It is a newtype wrapper over a record describing each of the four edges of the bounding box. This is a semigroup, where the binary operation produces the minimal bounding box that covers both inputs. That is, we take the minimum or maximum of the respective values, to get whichever maximizes the area covered:

```
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
                    }
```

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

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

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

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

Now, with our Monoid in hand, we can write another interpreter for Glyph, using Writer as our monad in the natural transformation:

```
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
-- and so on

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

With that, we get bounding boxes for free when constructing glyphs.

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

   First, some utility functions to generate and render GlyphPositions:

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

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

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

genThreeGlyphs :: Gen ThreeGlyphs
genThreeGlyphs = do
  l <- genGlyphPosition
  c <- genGlyphPosition
```

```
  r <- genGlyphPosition
  pure $ {l, c, r}
```

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

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

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

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

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

14.6 Putting it all together

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

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

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

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

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

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

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

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

NOTE these should be in discussion or something

14.7 Limitations/Performance

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

TODO cause: free monad

TODO potential solution: free applicative

15 Events

15.1 Notes

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

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

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

15.2 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.

In short, the system consists of four parts:

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

format it uses

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

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

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

15.3 Biodalliance

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

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

15.4 Cytoscape.js

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

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

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

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

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

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

```
exRec :: Record ( x :: Number, title :: String )
exRec = { x: 123.0, title: "hello" }
```

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

```
-- { label :: Type } is sugar for Record ( label :: Type )
exRec2 :: { title :: String }
exRec2 = { title: "another record" }

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

titleLength exRec == 5
titleLength exRec2 == 14
```

variants

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

```
_either = SProxy :: SProxy "either"
_or     = SProxy :: SProxy "or"
_nope   = SProxy :: SProxy "nope"

eitherOr :: forall r.
            Variant ( either :: Boolean, or :: Unit | r)
         -> String
eitherOr =
  default "neither!"
        # on _either (\b -> "either " <> show i)
        # on _or     (\_ -> "or unit")

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

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

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

eitherOr vEither == "either true"
eitherOr vOr     == "or unit"
eitherOr vNope   == "neither!"
```

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

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

(Footnote? It may be possible using type/value-level reflection/reification, as done in Functional Pearl: implicit configurations `http://www.cs.rutgers.edu/~ccshan/prepose/prepose.pdf`)

What I actually desired was a way to express events in an easy to configure way, while also guaranteeing correctness as far as possible, with good error reporting picking up the slack where necessary.

15.6 JSON zippers and stringy types

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

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

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

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

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

15.7 TrackSource and TrackSink

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

1. TrackSource

   The configuration needed for a TrackSource is a name, the JSON structure for the event to be produced, and the JSON structure of the event produced by the underlying track (e.g. Biodalliance).

   For this another library will be used, instead of purescript-foreign, namely purescript-argonaut.

   (a) Json decoding with Argonaut

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

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

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

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

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

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

   Into a JS object that looks like

```
{
  chr: "chr11",
```

```
  minPos: 1241230,
  maxPos: 1270230
}
```

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

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

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

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

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

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

We can grab the value at 'c' with

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

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

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

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

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

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

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

[1]DEFINITION NOT FOUND.

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

```
insideOut :: JCursor -> JCursor

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

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

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

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

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

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

```
-- This is just a nicer version of Tuple JCursor String
type Cursor = { cursor :: JCursor
              , name :: String
              }

type RawCursor = Cursor
type ValueCursor = Cursor


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

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

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

```
                    (a -> Either String b)
                 -> Array a
                 -> Either String (Array b)

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

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

2. TrackSink

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

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

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

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

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

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

   The following JS code defines a Biodalliance TrackSink (and is correctly typed):

```
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
                          }
                        ];
```

3. Running TrackSources and TrackSinks

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

   (a) 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 singleton list and the TrackSource type:

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

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

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

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

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

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

   Finally, a way to use a TrackSource is required. The following function uses a TrackSource to parse a provided JSON value:

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

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

   (b) TrackSink

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

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

Using a collection of ‘SinkConfigs‘ to produce a single ‘TrackSink‘ is not in itself complicated, but we do some validation, namely ensuring that there are not multiple handlers for a given event:

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

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

  pure $ foldMap makeTrackSink scs
```

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

To use a ‘TrackSink‘, we see if a handler for the provided event exists. If it does, we apply it to the contents of the event:

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

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

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

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

16 User interface

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

16.1 Biodalliance

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

WIP DOM actions 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.

TODO example

TODO Events NOTE: mainly covered in events.org

TODO example

WIP Problems 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.

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

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

16.2 Quick Halogen intro

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

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

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

1. Query algebras

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

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

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

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

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

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

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

‘ChildSlot‘ is a coproduct of the two child Slot *types* (Either2) of the child components; we can query the BD slot or the Cy.js slot at once, not both. ‘Either2‘ is a generalization of ‘Either‘ to a variable number of types, a convenience that makes it easy to change the number of slots, without more work than a type synonym. ‘ChildQuery‘ is a coproduct of the two child Query *functors* (Coproduct2).

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

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

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

– TODO: not sure, but it may even be impossible to do this; may not compile – (certainly doesn't compile when applied to Halogen)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
data UIBD.Message
  = SendBD Biodalliance

data UICy.Output
  = SendCy Cytoscape
```

   The main component can produce these:

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

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

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

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

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

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

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

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

16.3 The main application

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

– TODO: remove row blank when compiling with 0.12 – TODO: explain runHalogenAff

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

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

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

  Right (BrowserConfig config) -> do
```

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

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

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

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

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

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

```
io <- runUI component unit el'

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

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

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

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

Finally, we attach a callback to the Halogen component to listen for the reference to the BD instance, sent by the BD component upon creation. We then use the TrackSink and TrackSource configurations to hook BD up to the event system. Finally, we ask the main component to create the BD instance:

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

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

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

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

  _ -> pure $ Just unit

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

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

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

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

The Cytoscape.js code is analogous.

17 The Completed Product

Discussion

Conclusion

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

[6] 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.

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

[8] 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`.

[9] 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`.

[10] 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`.