Rhein – Data Propagation Library for Interactive Applications on the Web

Final Project Report

Author: Catalin Adelin Torge

Supervisor: Christian Urban

Student ID: K1763986

Programme: Computer Science BSc

June 22, 2022

Abstract

Interactive applications are everywhere, and the demand for this type of software is increasing at a rapid pace and yet, the development process for these applications has not changed much. The observer pattern and object-oriented programming are the two mainstream concepts that lie at the heart of the current software engineering trends. Research has shown that these methods tend to make the project's complexity increase exponentially and cause a lot of bugs.

In this project, we provide a new way of event handling using FRP abstractions. FRP is a new paradigm that provides new mechanisms to manage dependencies in your application. We provide a small data propagation library that targets web applications called Rhein .

Rhein aims to provide developers with a small library based on FRP abstractions to explore the benefits of FRP and further grow the community revolving around these ideas. The goal of this is to give enough proof and resources to understand and make it easy to explore this new paradigm.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Catalin Adelin Torge

June 22, 2022

Acknowledgements

Special thanks to my supervisor, Christian Urban, for his guidance and instruction throughout my project. It has been a great honor to work with him. The knowledge he has given me on various aspects of computer science has made me a better computer scientist. Thank you, Christian.

I would also like to thank the members of the Sodium forum (www.sodium.nz) for their suggestions and great answers on my questions while working on this project.

Contents

Chapter 1

Introduction

1.1 Brief

Interactive applications are everywhere, and the demand for this type of software is increasing on a rapid pace. The development process for this applications has not changed much, the observer pattern and object oriented programming are the two mainstream concepts that lie at the hearth of the current software engineering trends. The observer pattern in particular, provides great ways to handle event driven application logic, but at a high cost. Blackheath and Jones have discovered that the observer pattern is the cause of 6 common bugs in event-driven applications. Moreover, the nature of the observer paradigm makes program's complexity increase exponentially and this results in projects hitting the so called *complexity wall.*

1.2 Goal

This project aims to solve these issues by exploring new ways of event handling. FRP is a new paradigm that is based on functional and reactive programming. It provides new ways to handle events and manage the state of your application using abstractions like signals and behaviours that help you explicitly define the behaviours and logic in your application. The goal of the project is to create a small propagation library based on FRP abstractions to create interactive application on the web. This framework should constitute as a proof of concept that will provide enough evidence of the benefits FRP code brings to event-driven applications and hopefully attract more attention and build a community revolving around these ideas.

1.3 Report Structure

Chapter 1 provides a small introduction and the overall goals of this project.

Chapter 2 provides the background information and the research that has been made during the project. In this chapter, we state the issues of the observer pattern and introduce the new ideas that the FRP paradigm is based on. The end of this chapter is comprised of the motivation of the project and information about the programming language used to implement the propagation library.

Chapter 3 describes the features of Rhein . It covers the design and the concepts of the library.

Chapter 4 presents the UI Binding Library that is part of Rhein . Here we provide information about how we interpolate FRP code with user interfaces.

In Chapter 5 we talk about the journey that has led us to the final version of Rhein . Moreover, we discuss other FRP implementations as well.

Chapter 6 covers two applications that we provided to illustrate how we can use Rhein more concretely.

In Chapter 7 we discuss the professional and legal issues we encountered during the project.

Lastly, Chapter 8 and 9 cover the results, evaluation and the conclusion of the project.

Chapter 2

Background

2.1 User Interfaces and The Complexity Wall

On December 9th 1968, at the Computer Society's Fall Joint Computer Conference in San Francisco, a presentation that would later become known as ”The Mother of All Demos”, Douglas Engelbart and his team demonstrated almost all the fundamental elements of modern personal computing featuring text editing on a screen, his newly invented mouse, windows, graphics, hypertext links and a collaborative real-time editor. At that time, computers were room-sized machines perceived to outperform humans at computational tasks. Douglas introduced the idea that computers help humans to perform intellectual tasks, boosting human intelligence by becoming interactive assistants in everyone's daily work. The graphical user interface was born.

Today, building graphical user interfaces and using object-oriented languages have become mainstream. Unfortunately, programming user interfaces is still surprisingly burdensome. Event-driven programming and the observer pattern are the currently predominant style, and they have an unnatural tendency to quickly evolve into unstructured and difficult-to-maintain source code, often referred to as *spaghetti code* [2].

According to Blackheath and Jones, sooner or later many big software projects hit the so-called “complexity wall”. The complexities in the software seem innocent but quickly expand exponentially. Initially it is hard to notice them but with time, this mess becomes quite visible. There are typically several options when a project hits the wall:

- the project is put on hold
- the project is rewritten from scratch, and this implies a lot of expensive resources. This

option might lead to the same mistake, hitting the wall again in the future

- the projects undergo major refactoring, leading eventually to maintainable code

Refactoring, therefore, is the only solution. It is the primary tool to save a project that has hit the wall. It is best used as part of a methodology earlier in the development process to prevent the "hit" before it happens. Complex refactoring processes, such as applying big refactoring or removing design smells are difficult to perform in practice. "*The complexity of these processes is partly due to their heuristic nature and to the constraints imposed by preconditions on the applicability of the individual refactoring.*" [3].

2.2 Interactive Applications

Most applications are engineered with a programming model such as events or threads or a mix of the two. Events are discrete, asynchronous messages that are propagated around the program. A source of events are users who interact with the software for example, by emitting events like key presses or mouse clicks. Events are a more suitable model where a sequence of events is less obvious (e.g., mouse clicks from a user are unpredictable), especially when the interaction between the components of the application is more complex. Examples of software in this category are Graphical User Interfaces and video-games. Threads, on the other hand, model state transition as control flow and work best with I/O or any other situation when the state transitions fall into a clearly defined sequence. We can mention actors and generators here. [1, 4]

There is a continuously increasing demand for interactive applications which are driven by non-expert computer users. To be able to deal with the continuous user input and output, interactive applications require a great amount of engineering [5]. At the same time, to produce robust and efficient interactive software is challenging, and it is due to developers having to deal with asynchronous events, data consistency and propagating data and events through the application [6]. Most applications we write today are highly interactive and event-driven. Events are emitted either from the inside or outside of the application (e.g., mouse clicks are from the outside environment and a timer is from the inside). Take as an example an application that handles multiple input events (e.g., mouse clicks) that asynchronously arise from the GUI. It must react to all these events and it must continuously maintain interaction with the outside environment and process these and execute tasks in response (e.g., display data on the screen or update the state) [7].

Our programming models for these applications has not changed much. The particular paradigm developers use is still the observer pattern [8]. Many interactive applications use a mechanism of asynchronous callbacks (i.e., event-handlers) to handle reacting to external events. Using this mechanism is difficult and results in the problem known as Callback Hell [9]. This is difficult because we have to deal with unpredictable execution order that might modify the same data. Also, callbacks do not always return a value and this means that they have to perform side-effects[1] to the application state [11].

Why would we bother to propose a new way of even handling? An analysis in the Adobe's software presented in ”A Possible Future of Software Development” [12] on the status of current production systems we find that 1/3 of the code in Adobe's desktop applications is devoted to event handling and 1/2 of the bugs reported during a product cycle exist in this code. These numbers show the impact event handling has on the development process and shows that event handling produces a lot of bugs.

2.3 Stop Listening

Listeners, callbacks and the observer pattern they all refer to the same concept, and they are the predominant way of propagating events in software today. Blackheath and Jones provide a brief history of how dependency tracking was done before where, propagating some value through your application required getting the value and calling all the places that are going to use that value. This process gives the producer a dependency on its consumers. Therefore, to reuse code that produces events (e.g a list box) becomes a difficult task because the code is wired in the rest of the application. At the same time, the idea of reusing a component doesn't work well if it has to know in advance all consumers which will depend on it. All these issues have motivated the creation of the observer pattern and this is how it was born [1]. Using the observer pattern, to observe an event producer, at any time, you can register a new consumer (i.e listener) and from the moment when the listener is attached to the subject(another way to call the producer), it will be called back whenever an event occurs. To stop listening, you can deregister a listener at any time. This way, the observer pattern, inverts the natural dependency and the consumer now depends on the producer and not the other way around. This makes the program more modular by losing coupling between components.

Unfortunately, research shows that the observer pattern arises new problems, and it doesn't

[1] An operation, function or expression is said to have a side effect if it modifies some state variable value outside its local environment [10]

solve our problem with event handling, and it indirectly makes the development process of interactive applications harder and error-prone.

To further illustrate the problems with the observer pattern we will present and explain what Blackheath and Jones identified as "the six plagues of listeners" which are six sources of bugs with listeners [1]:

- *Unpredictable order* - The order in which events are received can depend on the order in which listeners were registered. This happens more in a complex network of listeners. This is important when you must ensure your application's GUI is glitch-free.
- *Missed first event* - It is difficult to guarantee that the first event is sent after you have registered a listener.
- *Messy state* - Callbacks push your code into a traditional state-machine style and it gets very complex and messy fast, especially when a class is listening to multiple event sources.
- *Threading issue* - Attempting to make listeners thread-safe can cause dead-locks and it is hard to guarantee that no more callbacks will be received after deregistering a listener.
- *Leaking callbacks* - Forgetting to deregister a listener can cause memory leaks. This can be prevented by holding weak references to the listeners.
- *Accidental recursion* - The order in which you update local state and notify listeners can be critical and it is prone to mistakes.

To give a different perspective and to show different issues with event handling code we will look at a simple example which Maier et al. have provided in their research paper about deprecating the observer pattern. It is about an application that draws a path from mouse movements and displaying it on the screen. Using this example, Maier et al. illustrated that the observer pattern encourages the violation of a great amount of important software engineering principles: encapsulation, resource management, separation of concerns, data consistency, and more [5].

2.4 What is FRP

Functional Reactive Programming (FRP) is a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming [13] which was first introduced by Conal Elliot and Hudak Paul in the paper "Functional Reactive

Animation" [14]. FRP languages define abstractions like signals and behaviours which are time-varying values. These concepts provide a more declarative way of modelling external events or internal changes by explicitly defining their behaviour. These abstractions manage dependencies automatically which gives developers the opportunity to express their application logic at a higher-level instead of dealing with the low-level implementation details of mutable state and callbacks that are required in the event-driven paradigm [15].

The original formulation of Functional Reactive Programming(FRP) can be found in the ICFP 97 paper Functional Reactive Animation by Conal Elliott and Paul Hudak [14] which introduces two abstractions: Behaviours(later called Signals) and Events. Behaviours are time-varying, continuous values while events are values that change at discrete points in time. Since 1997, FRP has taken many forms, from standalone languages to embedded libraries. Some of the ways it has been diversified are discrete vs. continuous semantics and how FRP systems can be changed dynamically. Some examples of FRP systems are Flapjax, Elm, Bacon.js, React4J.

In the previous sections, we have shown how programs can quickly go out of control and hit the complexity wall. Functional Reactive Programming is a new way of programming event-driven applications that deals with complexity in a specific way. Thanks to its functional nature, FRP enforces several mathematical properties, but the most important one is the principle of *compositionality* [2]. This property facilitates application components to be composed without unexpected side-effects.

2.4.1 Functional and Reactive Programming

FRP is an intersection of functional programming and reactive programming.

Functional Programming is a style or paradigm based on mathematical functions and avoids changing state and mutable data. It also implies the use of immutable data structures and emphasizes the principle of compositionality. Blackheath and Jones stated that compositionality is a powerful idea and it turns out to be why FRP can deal with complexity so efficiently.

On the other hand, *Reactive programming* is a declarative programming paradigm concerned with data streams and the propagation of change. In other words, applications written with this paradigm are usually event-based and they react in response to the input. These applications present a flow of data instead of the traditional flow of control. Thanks to these properties, reactive applications tend to be more modular due to the loose coupling between the application

[2] In mathematics, semantics, and philosophy of language, the principle of compositionality is the principle that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them. [16]

components.

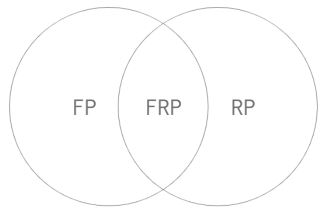


Figure 2.1: FRP is an intersection of Functional and Reactive programming.

There has been an increase in adoption for reactive frameworks, especially since the *Reactive Manifesto* [17] has been published in 2014. A bunch of libraries have been developed for all major programming languages and to a lesser extent, Google's *AngularJS* [18] and Facebook's *ReactJS* [19] have also been inspired by the reactive paradigm. These libraries provide an enhanced data binding which is a general technique that binds data sources from the provider and consumer together synchronising them. A more recent library, *Combine* is another proof that reactive programming is getting more attention from the developer community. Combine is a new framework by Apple introduced at the company's annual developer conference WWDC 2019 [20]. The framework provides a declarative Swift API for processing values over time. Apple defines the framework as a way to customize the handling of asynchronous events by combining event-processing operators. This new framework can be compared to *ReactiveCocoa* [21] and *RxSwift* [22] which is a library that implements the Reactive Manifesto specifications for the Swift language.

All these frameworks belong to the family of FRP due to their nature of processing values over time but they are not pure FRP as defined by Elliott and Hudak. While these libraries are great, they are most of the times called FRP while that's not quite right. As it is stated in the ReactiveX documentation, these libraries are sometimes called "functional reactive programming" but this is a misnomer. For example, ReactiveX may be functional, and it may be reactive, but "functional reactive programming is a different animal" [23]. The main point of difference between reactive libraries such as Rx and FRP is that these libraries mostly just look at events and not behaviours. Events, as defined at the beginning of section 2.4 are discrete values that are emitted over time, such as mouse clicks. Behaviours are continuous values that always have a current value, such as a mouse position. A mouse click itself doesn't have a value – it is just an event that gets fired every time the user clicks somewhere. By contrast, a mouse position always has a current value – but it doesn't get fired at certain points in time. At the same time, these libraries don't have a denotational semantic specification which is a required property for true FRP systems. A more concrete definition of true FRP will be presented in section 2.4.2 where we will give examples and see the differences between behaviours and events. Nonetheless, it's great to see that these new paradigms are getting more popular, and there's no doubt more libraries and languages will follow both reactive and FRP.

2.4.2 The true FRP definition

As we motioned in the previous section, FRP has taken a lot of paths since it has been proposed by Elliott and Hudak. A more concrete definition is given by Conal Eliott in a Stack Overflow answer to a question regarding the true definition/specification of FRP. Conal's answer is the following: "*I'm glad you're starting by asking about a specification rather than implementation first. There are a lot of ideas floating around about what FRP is. For me it's always been two things: (a) denotative and (b) temporally continuous. Many folks drop both of these properties and identify FRP with various implementation notions, all of which are beside the point in my perspective. To reduce confusion, I would like to see the term "functional reactive programming" replaced by the more accurate & descriptive "denotative, continuous-time programming" (DCTP). By "denotative", I mean founded on a precise, simple, implementation-independent, compositional semantics that exactly specifies the meaning of each type and building block. The compositional nature of the semantics then determines the meaning of all type-correct combinations of the building blocks.* " [24]. A true FRP system has to be specified using denotational semantics.

Denotational semantics is a mathematical expression of the formal meaning of a programming language. For an FRP system, it provides both a formal specification of the system and a proof that the important property of compositionality holds for all building blocks in all cases, which is also an important property in software design [25]. One of the main goals of denotational semantics is to specify programming language constructs in as abstract and implementation-independent way as possible.

The mechanism of *Continuous time* is to update a behaviour representing time before passing external events into the FRP system. Externally, you're saying "Please sample the model at time `t`", but within the model you can think of time as varying continuously. This makes it easy for FRP to simulate physics in a natural way.

Events and Behaviours are both first-class values in FRP, and there are a rich set of combinators (operators) that the programmer can use to compose new behaviours and events from existing ones. A FRP program is just a set of mutually-recursive behaviours and events each of them build up from non time-varying values and or other behaviours and events.

2.4.3 Motivation

After reviewing a few FRP papers we find that they describe systems that worked brilliantly on their own, and have some great properties. Yet, they require the entire program to be written in an ambiguous variant of an ambiguous language. Inter-operability with existing languages or paradigms was ignored at all, which results in being impossible to incrementally introduce FRP into an existing code-base.

At the same time, the reviewed literature shows an amazing new FRP ecosystem that we could adapt but often with the condition to port all our code to either Haskell or Scheme since most of the proposed solutions have been implemented in the mentioned languages.

This project goes on stage further by proposing a small change propagation library for defining better user interfaces. It discusses the different variations of FRP and how we can we use abstractions like Signals to explicitly define the behaviours in our programs and focus more on the ”what” rather than on the ”how”.

2.5 Scala

Scala is a general-purpose programming language providing support for functional programming and a strong static type system.

Scala combines object-oriented and functional programming in one concise, high-level language. Scala’s static types help avoid bugs in complex applications, and its JVM and JavaScript runtimes let you build high-performance systems with easy access to huge ecosystems of libraries.

One of the main reasons that this language has been selected to implement this project is the functional nature of the language. This is needed as FRP is highly based on functional programming features. Moreover, functions in Scala are first-class object and we can compose them with guaranteed type safety. This is another big advantage that will ease the implementation phase.

2.5.1 Scala Build Tool (sbt) & Metals

Scala can be compiled using its own compiler, but it is difficult to manage a complex project without a build tool. sbt is an open-source build tool for Scala and Java projects, similar to Apache's Maven and Ant. It provides native support for compiling Scala code and integrating with many Scala test frameworks. sbt has interesting features like defining tasks in Scala which you can run in parallel from sbt's interactive shell. It also provides continuous compilation, testing, deployment and dependency management using Apache Ivy (which supports Maven-format repositories).

Scala only provides native support for InteliJ IDE. In order to benefit from features like *goto definition* and *completions* with other IDEs (in this project we decided to use VS Code) we must use Metals.

Metals is a Scala language server with rich IDE features. It provides features like compile on file save and checking errors from the build tool inside the editor. There is no need for switching focus to the console. The build tools sbt, Gradle, Maven and Mill are supported thanks to Bloop. Hot incremental compilation in the Bloop build server ensures compile errors appear as quickly as possible. Goto definition, completions, hover (aka. type at point), signature help (aka. parameter hints) are just a few features this tool offers.

Chapter 3

Rhein – Concepts & Design

3.1 Definition & Inspiration

Rhein is a data-propagation library based on Functional Reactive Programming abstractions such as Events and Behaviours that helps you to develop interactive applications using a conceptual-declarative approach that brings numerous benefits to the quality of the applications and also solves several problems the mainstream methods of development of this type of software produce.

Rhein is based on multiple existing FRP implementations. The main source of inspiration when implementing and designing the library was Sodium [26], an FRP library by Stephen Blackheath. Sodium is a push-based FRP system where, behind the scenes the observer pattern is used to maintain dependencies. Sodium is a system with denotational semantics that has all characteristics of a true FRP system. Scala.rx [27] and Scala.React [5] are two other sources of inspiration when designing Rhein . Scala.React is the implementation of the popular paper by Maier et al. "Deprecating the Observer Pattern" which introduces a data propagation library in Scala. It uses delimited continuations and other interesting concepts that also inspired Haoyi in his own system Scala.rx. Although, Scala.React is a true FRP system, Scala.Rx lacks continuous time and denotational semantics, two properties that are required by a true FRP system. Concrete ideas and concepts we have taken from these libraries will be mentioned in the continuation of this chapter.

3.2 Structure

Rhein consists of two main components: the FRP Engine and the UI Binding library which is based on Scalatags[1]. Both components work seamlessly together and they provide all fundamental components that are required to build an interactive web application. The benefits of separating the two are to facilitate further improvements and further work to creating other UI Binding libraries that could target desktop applications (using JavaFX) or even mobile. Therefore, in the future we can see FRP used not only on the web but also on other platforms as well. Moreover, the FRP Engine doesn't depend on the UI Binding Library nor Scalatags and it can be used for state management or any other event handling and processing and not just UI code.

Rhein is based on declarative programming ideas and therefore while developing using this framework you will find yourself more concerned with what your app should do and less with how it should do it. Writing code in Rhein requires the developer to think conceptually and not operationally. Because Rhein is based on FRP concepts, the application logic is a flow of data. Data and information flow into your logic through Events and Behaviours. Data flow towards the output side, and the section in the middle is also a flow of data. Data flows from input to output. FRP is fundamentally a declarative description of the output in terms of the input. Rhein will make you stay conceptual and think at the level of relationship between components and not the mechanics of their interaction.

3.3 The FRP Life Cycle in Rhein

An FRP system usually has two stages in terms of its life cycle. Figure 3.1 shows the mechanics of how FRP code is executed in Rhein . In most FRP systems as well as in Rhein , this process happens at runtime, and consists of two stages:

- Initialization: Typically during program setup, FRP code statements are converted into a directed graph in memory representing the dependency relations in our program.
- Running: For the rest of the program execution, we feed values and turn the FRP engine produces the output.

One of the main tasks of the FRP engine is to make sure information is processed in the order specified by the dependencies in the directed graph that is stored in memory. In a spreadsheet

[1]Scalatags is a small, fast XML/HTML/CSS construction library for Scala that takes fragments in plain Scala code by Haoyi. It can be found at http://www.lihaoyi.com/scalatags/

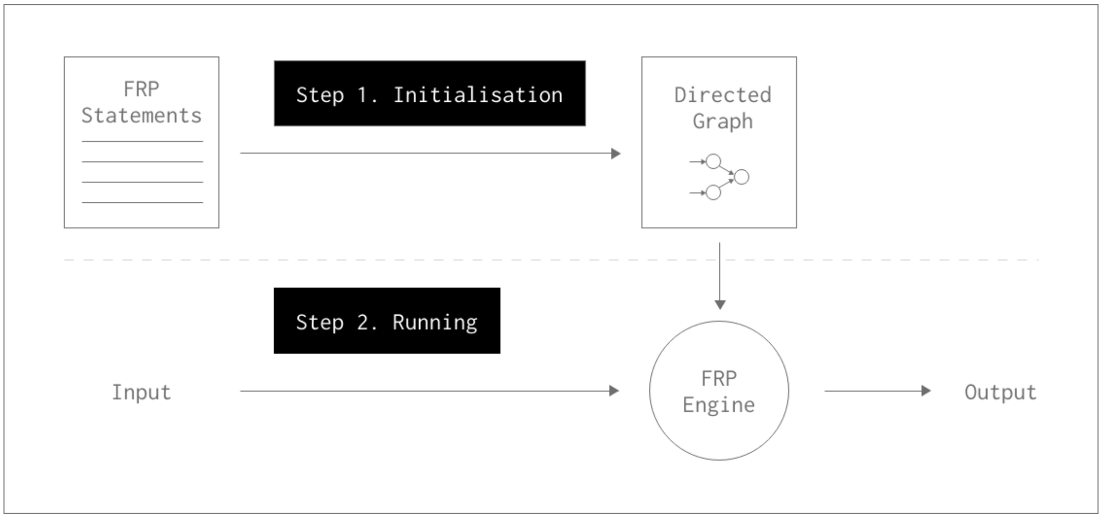


Figure 3.1: Life Cycle of an FRP system.

application this is referred as to ”natural order recalculation”[2]. We will refer to this as the “correct order” to distinguish it from any other types of orders, which could give the wrong result.

The separation between the initialisation and running stages is similar to the way GUI libraries work. In the Java GUI Swing library, you construct the visual widgets first and afterwards an event loop handles events from the user.

To illustrate this practically, we will have a look at a simple flight booking application example which can be found in Listing 1 where the user can select a departure and a return date. While the user provides the input, business logic continuously makes decisions about whether the input is valid. The business logic is simple and it looks like this `valid = true if departure <= date`. In figure 3.2 we can see a conceptual view of our program.

```
DatePicker depElem = new DatePicker();
DatePicker retElem = new DatePicker();
Event[Date] dep = depElem.event;
Event[Date] ret = retElem.event;
Behaviour[Boolean] valid = dep.lift(ret, (d, r) -> d.compareTo(r) <= 0);
Button ok = new Button("OK", valid);
```

Listing 1: Flight Booking Example

Note that in Listing 1 we omitted to explain a very important aspect of our program: how events are being fired inside the `DatePicker` objects, that is how does FRP code interpolate

[2]The term ”natural ordering” in spreadsheets applications is a special case of a more general idea called topological sorting, in which a set of objects with dependencies are sorted in a way such that each object is processed only after the objects on which it depends

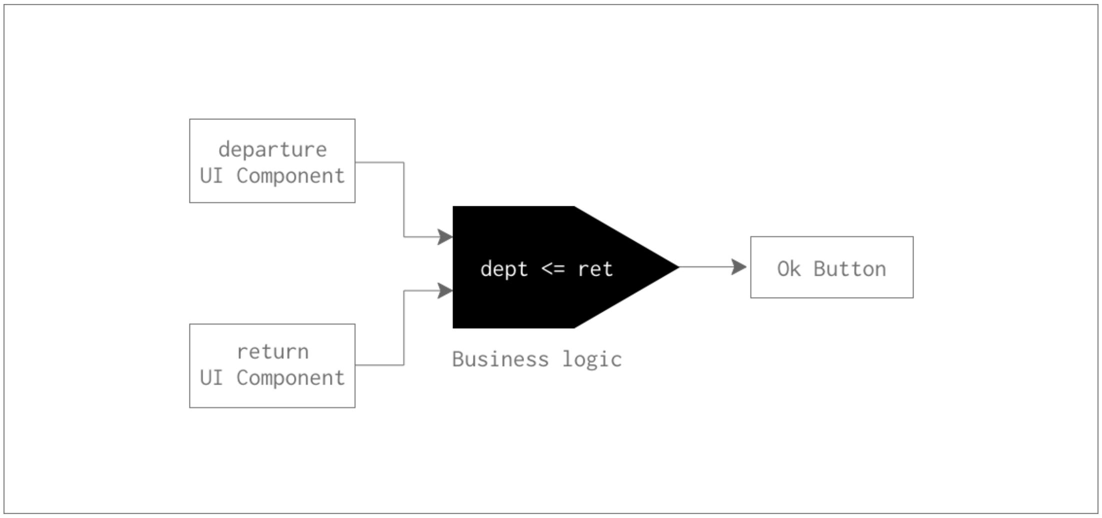


Figure 3.2: Conceptual view of flight booking application.

with operational code. This mechanism will be explained more in the following chapter, but for now, we want to focus on the dependency relation and the conceptual view of the program.

During the initialisation stage the FRP engine executes the statement at line 5 in Listing 1 which describes the relationship between our three components. Once initialisation is over, we enter in the running stage and the system will process incoming events from the user. The engine's job is to maintain the relationship we expressed ensuring that value of `valid` is always up to date.

Both spreadsheet applications and FRP share a number of key features, and in fact FRP works the same way as a spreadsheet software does. Moreover, we can write our flight booking program in a spreadsheet application but we can actually write arbitrarily complex application logic using the style of a spreadsheet. FRP allows you to do this as well, but it requires a different way of thinking.

3.3.1 Paradigm shift

A paradigm is a way of thinking, a philosophical framework, world view or a frame of reference and usually is applied to a particular area of knowledge. In order to understand FRP in more detail, one must first get used to the way of thinking in FRP. There are multiple components in a program, and some depend on each other. Software is expressed as a sequence of steps and in the same way some components of a software depend on each other, some steps depend on previous steps. Blackheath and Jones give a nice example illustrating dependency. They analyse two processes: `washing face` and `brush hair`. There is no dependency between the two and they can be executed in any order. By contrast, if we take as an example the processes `open silo door` and `fire missile`, in this case the dependency implied by the sequence is critical. This is a popular example given in the functional programming community.

Given a conceptual diagram like the one we created for the flight booking application in figure 3.2, we can extract the dependency relations easily by removing the unnecessary information like the business logic and reverse the data-flow arrows. We can note that `valid` is dependent on two values: `departure` and `return`. The FRP engine learns all these relationships so it can automatically determine the dependencies which guarantee the correct sequence.

Real programs have much more complex sequence-dependent code compared to our example, and one of the biggest problems with this code arises when it needs to be modified. Making sure something happens earlier or later requires a full understanding of the implicit dependency in the existing sequences. In FRP you express dependencies directly and it becomes easy to remove or to add new dependencies because the sequence is automatically updated. By contrast, in listener-based event handling, dependencies are still expressed but it is still difficult to maintain a reliable sequence. The order in which the sequence is processed depends on when the code propagates events and on the order in which listeners are attached.

Thinking declarative, what the program is not what it does could arguably be one of the end goals of FRP. As developers, we spend a lot of time translating problems that are formulated in terms of dependencies into sequences. In other words, as Blackheath and Jones have concluded, we are working in the "machine space" rather than the "problem space". In FRP the sequence is derived from the dependency relation and that gives developers the opportunity to focus more on "what" and a lot less on "how". This style is referred to as *declarative programming*, where you tell the machine what the program is, not what it does by directly stating information and relationships between them.

3.4 Data types

Rhein has two main data types: `Event` and `Behaviour` and they correspond to the two FRP abstractions we've mentioned so far. `Event` represents a stream of events whereas `Behaviour` represents a value that changes over time. In Rhein we also have 7 basic operations called primitives which help us convert events to behaviours, combine them and create other more complex operations based on these primitives.

3.4.1 Event Class

Event is a stream of discrete events also known in other FRP systems as Stream, Observable or Signal. When an event propagates through our event data type (stream) we say that an event has been fired. When this happens, an event or message is propagated from one part of the program to another. The message usually contains a value, often referred to as the payload

In Rhein we created a class called `Event` which is parameterized by a type of the value that it propagates, the payload. For example, to create a stream of click events you do:

```scala
val clicks: Event[Unit] = new Event()
```

Listing 2: Creating an Event

Figure 3.3: Visual representation of an event. Notice how the firings/event occurrences don't have a value in this case because they represent click events. In general, an event can have a payload of any type.

3.4.2 Behaviour Class

Behaviour is a container for a value that changes over time also known in other FRP systems as Cell, Signal. The main difference between event and behaviour is that events fire at discrete times and only have values at the moment they fire. By contrast, a behaviour always has a value that can be sampled at any time. In Rhein behaviours model state and event model state changes. Behaviours can be used to model the position of a mouse on the screen, time, a state with the current todos that are still to be completed. To create a behaviour you do:

```scala
val myBehaviour: Behaviour[String] = new Behaviour(Some("String"))
```

Listing 3: Creating a Behaviour

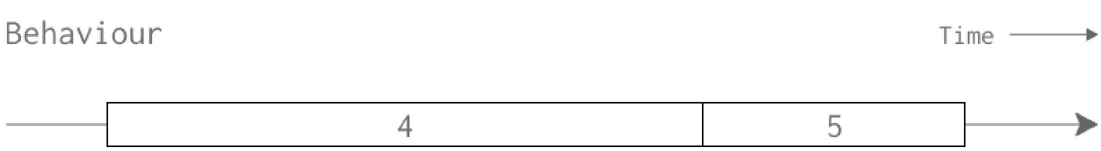


Figure 3.4: Visual representation of a behaviour. Notice how a behaviour always has a value.

To illustrate the use case of a behaviour, let's take a look at a small example. A text label that shows the current text of a text field. When the the text is changed, the label will reflect the changes.

```
val textField: TextField = new TextField("Hello!")
val label: Label = new Label(
    textfield.text.map((text => text.toString.reverse))
)
```

Listing 4: Example of what a behaviour can be used for.

We don't actually see any behaviours in this example and that's because the `TextField` object that is part of the UI Binding Library is injected with a behaviour under the hood. The property `text` of the `textField` is the "state" of the text field and it's passed to the label and has a type of `Behaviour[String]`. Note that we're also doing a transformation on the text using the `map` primitive. The label will actually reflect the text in the mirror.

3.5 The map primitive

The `map` primitive is used to convert a stream of events of type `A` into a stream of events of type `B` by passing a function as an argument that does the transformation. The `map` primitive works on both abstractions and it is semantically equivalent to the `map` operator that we can apply to collections(i.e. lists).

A simple example of the `map` primitive in action is transforming a stream of click events that are of type `Unit` into a stream of type `String`. If we think about a simple application where we have a text input and a clear button, we can write this using a stream and the `map` primitive.

```
val button: Button = new Button("Clear")
val eventClear: Event[String] = button.eventClicked.map(u => "")
val textField: TextField = new TextField(eventClear, "Initial text")
// rendering elements in the DOM
```

Listing 5: Clearing text in a text field using map

There are three things happening in Listing 5:

1. An event is generated and pushed into an event stream named `eventClicked`. A button click is not associated with any value and therefore a click event contains nothing (the payload is of type `Unit`)

2. This event propagates to a `map` operation that transforms the `Unit` type into a value of `""` (empty string). This `map` produces a new event stream that we called `eventClear`.

3. The event fired in `eventClear` propagates to the text field and changes its content according to the payload received which in this case is the empty string.

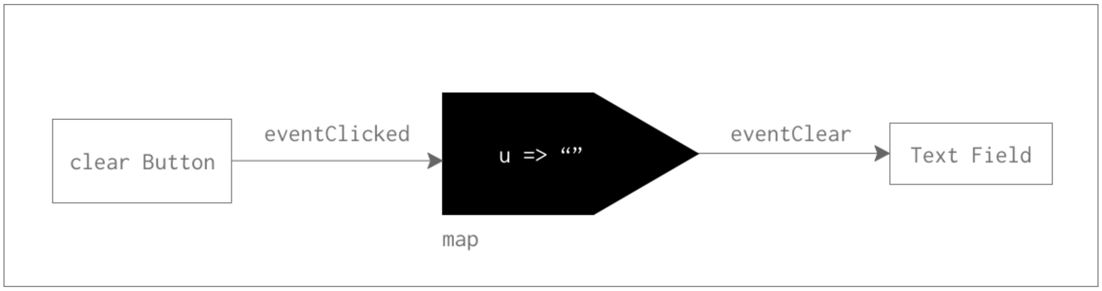


Figure 3.5: Conceptual view of map primitive as seen in the example in Listing 5

In the example above we convert a button click into a text field update. We do this by passing a function to `map` that convert `Unit` to `String`. Whenever the `eventClicked` fires, the `eventClear` fires at the same time. To perform the conversion, `map` executes the code inside the function provided each time `eventClicked` fires.

3.6 The merge primitive

The `merge` primitive puts the event firings from two event streams together into a single stream. This function is semantically equivalent to the one we apply to collections (i.e. lists). The name `merge` is universal but in other mathematical terms can be found as ”union”.

The two input event streams and the output must all have the same type. `merge` gives you an event stream such that if either of the input event stream fires, an event will be propagated to the output event stream at the same time.

To give an example that better illustrates the `merge` primitive we will look at a simple application that works as follows: We have a text field similar to one we find in a chat application and two buttons that help the user to quickly reply a message `"Bye!"` or `"Hello!"`. If the user clicks the `byeButton` the text `"Bye"` will be inserted in the text field and `"Hello!"` if the other button is clicked.

```scala
val byeButton: Button = new Button("Bye!")
val helloButton: Button = new Button("Hello!")
val merged: Event[String] = Event.merge(
    byeButton.eventClicked.map(u => "Bye!"),
    helloButton.eventClicked.map(u => "Hello!")
)
val textField: TextField = new TextField(merged, "")
```

Listing 6: Inserting a message in a chat application corresponding to a button click.

Listing 6 gives the code for the simple application. `TextField` only takes one event stream as input, therefore we need to merge our two streams from the two buttons. Note that we are also applying a transformation on the button event streams, that is the click event of type `Unit` is transformed into a type `String` with a given value. The text inside the `textField` object will initially be empty. As soon as we click on either of the buttons, the text would be changed accordingly.

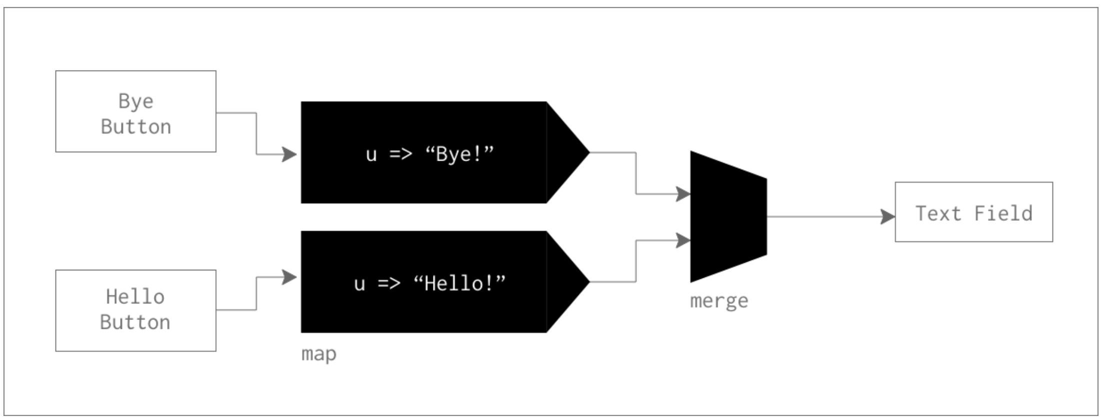


Figure 3.6: Conceptual view of merge primitive.

In Rhein and also some FRP systems, processing events takes place in a transactional context and it has the same behaviour as a transaction used in a database. Transactions in Rhein are inspired by the Sodium library. In a similar manner, a transaction is automatically started whenever an input value is pushed into an event stream or behaviour. Any state changes that occur as a result of that input are performed within the same transaction.

3.6.1 Simultaneous events

Simultaneous events are two or more event occurrences in an event stream that occur in the same transaction. In Rhein they are truly simultaneous because the order in which they were processed or fired cannot be detected. In the example in Listing 6, simultaneous events cannot happen, therefore `merge` at line 3 will never encounter the situation where two events occur in the same transaction. This is because of the way the UI Binding library is designed; the `Button` object creates a new transaction whenever it emits an event.

Even though it is not possible to have simultaneous in our example, we can still encounter them. Blackheath and Jones have provided a great example of a situation where simultaneous events occur.

They referred to a GUI application for drawing diagrams in which graphical elements can be selected or deselected. The rules are the following:

- If the user clicks on an item, it becomes selected.
- If an item is selected and the user clicks elsewhere it gets deselected.

In figure 3.7 we can see a sequence of actions that a user might follow:

- At time 1, nothing is selected and the user is ready to click the triangle.
- At time 2, The user clicks on the triangle and it gets highlighted.
- At time 3, the user gets ready to click the rectangle.

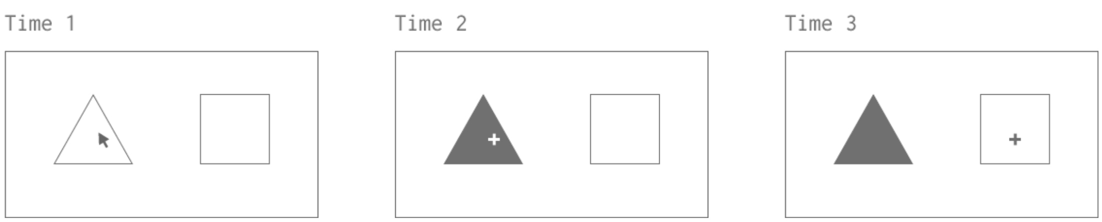


Figure 3.7: Sequence of actions that a user might follow

At this point, in time 4, a single click will generate two simultaneous events: deselecting the triangle and selecting the rectangle. These event streams would most probably be merged at some point in the application. Because the two events originated from the same click event they are simultaneous. All three of them are conceptually simultaneous because their order relative to each other cannot be determined. This helps us fix the *unpredictable order* plague from chapter 2.

As a policy, merging simultaneous events should be forced because that way events simultaneity is guaranteed [1]. Rhein doesn't provide an additional merge primitive for combining simultaneous events, we just take the first event and the rest (simultaneous) events are dropped. This could be further improved in future versions by creating another version of the `merge` primitive that takes as a parameter a function that is responsible with combining simultaneous events.

This property of forcing to combine simultaneous events is important because this makes our code more composable. Handling how simultaneous events are combined is a matter of how this process is handled locally in the context of the `merge` primitive when it's used and not in other parts of the program. This is important for reducing bugs.

3.7 The hold primitive

The hold primitive converts an event stream into a behaviour in such way that the behaviour's value is that of the most recent event received. Other names in other FRP systems for this primitive are stepper or toProperty.

Note that behaviours are how we model state in FRP and it is the only mechanism for doing so. Behaviours are like memory. `hold` allows storing an event stream's value so it can be retrieved later.

To illustrate how this primitive works we will take a look at a simple application: There are two buttons and a label. We'd like to display a value according to which button is clicked.

```
val buttonYes: Button = new Button("Yes")
val buttonNo: Button = new Button("No")
val merged: Event[String] = Event.merge(
    buttonYes.eventClicked.map(_ => "Yes"),
    buttonNo.eventClicked.map(_ => "No")
)
val label: Label = new Label(merged.hold("Please select an option."))
```

Listing 7: Reflecting how hold primitive works as acting as a state for the current button pressed.

In our example, we are using two other primitives that we covered so far: `map` and `merge`. `map` is used to convert the click event that has no value associated to it into a string. The two button click events correspond to two string values `"Yes"` and `"No"`. `merge` is also used to put the two streams of events into a single one. `hold` is used to convert this stream into a behaviour that we feed into the label. Therefore, whenever a button is clicked the behaviour will reflect the correspondent value. Also, note how `hold` takes an initial value; that is because behaviours always have a value.

3.8 The snapshot primitive

We have seen how a behaviour can hold a value that we type in a text field and it updates constantly as we type. In a real application, this might be a bit too distracting for the user. There is a different way of reading text from a text field using the `snapshot` primitive.

We are going to illustrate the mechanics of the `snapshot` primitive using a translate application. The application has three components: a text field where the user inserts the text to be translated, a button that triggers the translation and a label showing the end result. We translate the text from English to mock Latin[3].

```scala
val translateButton: Button = new Button("Translate")
val english: TextField = new TextField("English...")
val translated: Event[String] = translateButton
    .eventClicked
    .snapshot(english.text,
        (u, txt: String) => txt.trim.replaceAll("|$", "us").trim
    )
val latin: Label = new Label(translated.hold(""))
```

Listing 8: Translating English to mock Latin using the snapshot primitive

Analysing Listing 8, the value in the `english` text field is sampled when the translate button is clicked and the output is the result of the translate function provided in the `snapshot` primitive that is applied to the two input values: the event stream of clicks `eventClicked` from the `translateButton` and the value sampled from the text field's internal behaviour `english.text`. We can see a conceptual view of this application in Figure 3.8.

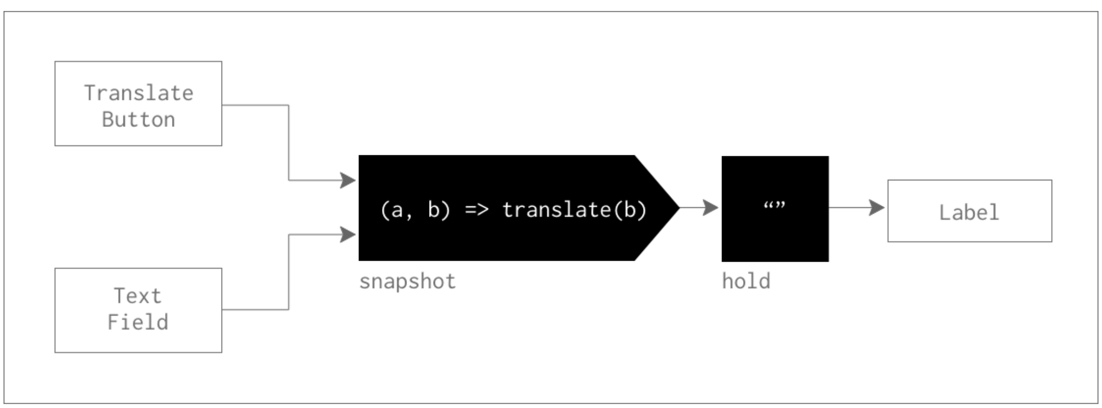


Figure 3.8: Conceptual view of the dependency relation of our components and the operations applied.

[3]To create mock Latin, we add ”us” to the end of each word. This is not a real language, and it’s just for illustration purposes

The `snapshot` primitive captures the value of a behaviour at the time when an event stream fires, and then it can combine the payload from the event stream and the one from the behaviour together with a supplied function. In our example, we discard the click event and only process the text received from the behaviour. This is illustrated in Figure 3.9.

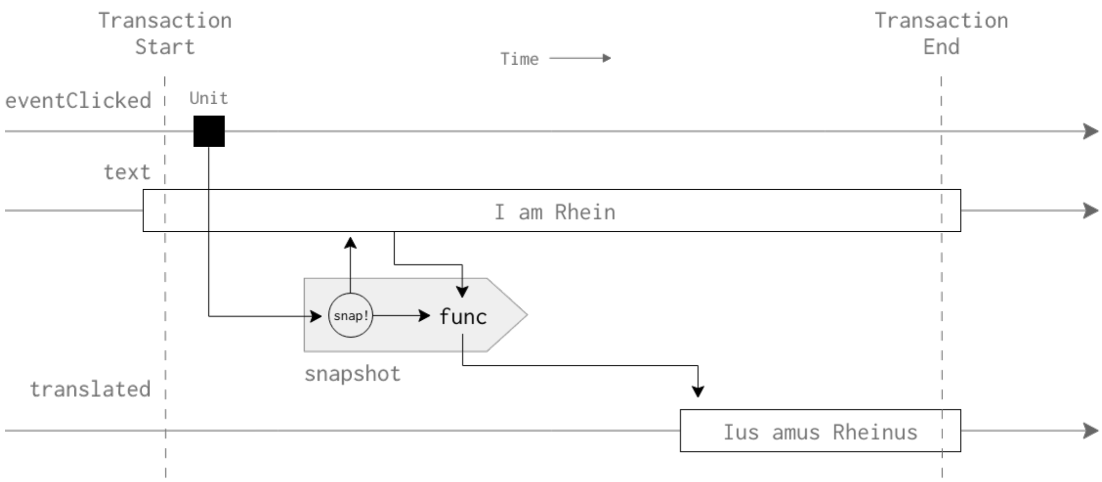


Figure 3.9: Time sequence of execution of a snapshot operation.

3.9 The filter primitive

The `filter` primitive is used to let event stream values through the pipe only sometimes. This is a general functional programming concept, and this name is used universally in FRP systems. It is semantically equivalent to the filter combinator that is applied to collections(i.e. lists).

The way `filter` works, is by applying the primitive to an event stream and provide a function that gives which values are allowed to pass through. In case you need to filter based on some state(i.e. behaviours) it is required to apply `snapshot` to the behaviour first and then filter the output. To filter out negative numbers on a stream you write:

```
val positives: Behaviour[Integer] = numbers.filter(n => n >= 0).hold(0)
```

If the value of `numbers` is greater than 0 it is let through. If not, it is discarded, and no update is received by `hold`.

3.10 The lift primitive

The `lift` primitive allows you to combine two or more behaviours into one using a specified combining function. Rhein only provides a way to combine two behaviours at a time at this point. This could be improved in further versions.

To illustrate this we will present a small calculator application that adds two numbers together. The application has two text fields and one label that holds the result of the addition. Listing 9 provides the code for this.

```scala
val textFieldA: TextField = new TextField("0")
val textFieldB: TextField = new TextField("0")
val a: Behaviour[Int] = textFieldA.text.map(t => t.toInt)
val b: Behaviour[Int] = textFieldB.text.map(t => t.toInt)
def add(a: Int, b: Int): Int = a + b
val lifted = a.lift(b, (p, q) => add(p, q))
val res: Label = new Label(lifted.map(x => x.toString))
```

Listing 9: Adding two numbers together using lift

Figure 3.10 shows the conceptual view. We convert the input text field values to integers, add them using the `lift` primitive and then converting back the result to string and feed it into a label.

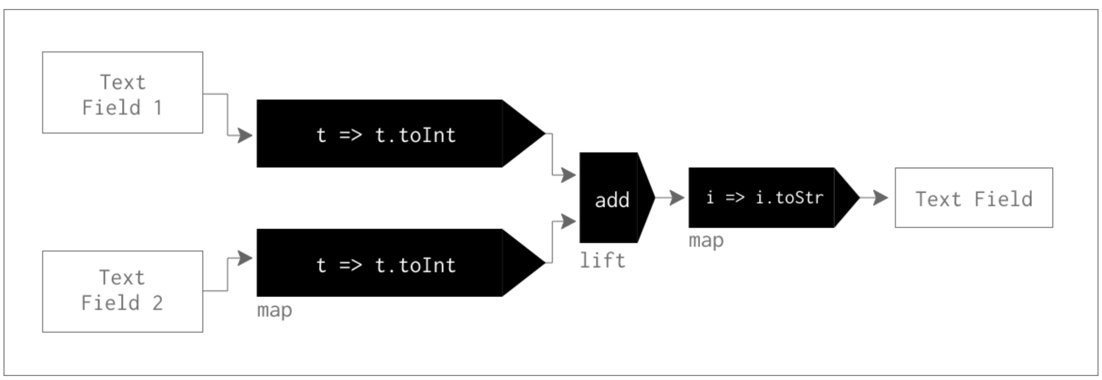


Figure 3.10: Conceptual representation of the calculator application.

The `lift` primitive is similar to the `map` primitive that works on behaviours, except that it takes two or more behaviours as input instead of one. Additionally, `lift` takes a function as its last argument with multiple input arguments equal to the number and types of the lifted behaviours.

Lift is a functional programming term that is responsible to make a function that operates on values into a function that operates on some type of container of those values. The name

”lift” comes from the fact that you are ”lifting” a function that operates on values into the world of behaviours. `Behaviour` is the container type in our example, but you can also lift into other container types such as `Optional`.

3.11 Loops and Accumulators

Before we talk about loops or accumulators, it is important to understand why we need them in Rhein . The idea of loops is specific to the Sodium library where we got inspired from, whereas the accumulator (sometimes referred to as scan) is a common primitive that we find in other FRP systems.

To introduce loops, we will provide an example that will make them easier to understand. We will show how we can implement a small spinner. The accumulated result is stored in a behaviour and we use the UI Binding Library to display it. We also have two buttons that change this value. Before we provide the complete code that implements this small application, it is required we understand the concept of forward referencing.

The first step in making this application we need to define two event streams and merge them.

```
// UI Buttons defined here...
val plus: Event[Int] = plusButton.eventClicked.map(u => 1)
val minus: Event[Int] = minusButton.eventClicked.map(u => -1)
val merged = Event.merge(plus, minus)
```

Listing 10: First step of the implementing a spinner in Rhein

After this, all we have to do is to accumulate the merged events into a behaviour.

```
val updates = merged.snapshot(state, (delta, state_) => {
    delta + state_
})
val state: Behaviour[Int] = updates.hold(0)
```

Listing 11: The second step of implementing a spinner in Rhein

Unfortunately, this code will not compile because we are defining `state` in terms of itself: `state` depends on `updates` and `updates` depends on `state`. There is a loop!

Value Loop, in functional programming is a value defined directly or through other variables in terms of itself. To make this possible in Rhein , we provide two extensions of the Behaviour and Event class: `BehaviourLoop` and `EventLoop`. These two classes provide a way to get around with the forward referencing issue.

`BehaviourLoop` and `EventLoop` are immutable variables that you can assign once using the `loop()` method that they provide, which you can reference before it is assigned. Please find the final code for the spinner application in Listing 12.

```
 // UI Buttons defined here...
val plus: Event[Int] = plusButton.eventClicked.map(u => 1)
val minus: Event[Int] = minusButton.eventClicked.map(u => -1)
val merged = Event.merge(plus, minus)

val state: BehaviourLoop[Int] = new BehaviourLoop()
val updates = merged.snapshot(state, (delta, state_) => {
    delta + state_
})
state.loop(updates.hold(0))
```

Listing 12: Final implementation of spinner in Rhein

An *Accumulator* represents a state that is updated by combining new information with the existing state. As we mentioned above, FRP systems usually provide a specific primitive for this which is sometimes called ”scan”. Its mechanism can be simulated using `hold`, `snapshot` and `loops`, and therefore we didn’t include an explicit primitive for this. This could be added in future versions of Rhein .

3.12 Interacting with the external environment

On `Event` streams we can attach listeners and we can also send/emit events down the stream (you can also think of this as a pipe where we send messages from one side to another). These two operations belong to the operational side of things and should not be considered FRP. Rhein provides a UI Binding library that uses these operational functions to facilitate the inter-operability with general languages and more concretely with your existing code or application.

To illustrate how we use operational code inside Rhein we will take a look at the implementation of the `Event` type and the `map` primitive.

There are two event types in Rhein `Event` and `EventSink`. The first one is the general type which should be used in the FRP part of your application. `EventSink` is an extension of the `Event` class providing an additional method called `send` that is responsible with emitting events. When this method is called a new transaction is created and all the nodes/events that

```scala
val eventSink: EventSink[String] = new EventSink()
val listener: Listener = eventSink.listen(payload => {
    println("I have received a new value: " + payload)
})
eventSink.send("First value")
eventSink.send("Second value")

/*
Output:
>I have received a new value: First Value
>I have received a new value: Second Value
*/
```

Listing 13: Emitting events in Rhein

are dependent or listening will be updated or called (in the case of listeners, usually this is a piece of code similar to a lambda function that receives a payload that represents the value emitted in the event). Please refer to Listing 13 for a concrete example.

Each event has a list of listeners and a list of finalizers (nodes that are about to be closed when this event dies / is killed). The list of listeners gets increased when we attach a new listener to an event. On the other hand, we cannot attach listeners to `Behaviour` objects. This is because behaviours in FRP always have a value. Instead, we can sample them at any time using the `sample` method that is available on `Behaviour` objects. Nonetheless, the `Behaviour` class has an event stream injected that represents the changes of the behaviour value over time. Whenever the event inside emits the new value, the behaviour value changes. We can attach a listener to that internal `Event` object if we need to, but this also belongs to the operational part. To conclude, we use operational code (`listen()` and `send()`) to make it possible to create more complex event streams. These two methods are frequently used in the implementation of the 7 primitives.

To illustrate this further, in the implementation of the `map` primitive which is presented in Listing 14, we notice that `map` uses an `EventSink` internally (line 2). As we mentioned before, when we map one event of type `A` to another event of type `B` we create a new event stream that depends on the one we applied `map` on. The new event listens to the current event that emits a payload down the stream to the new event and the new event applies the transformation function supplied as a parameter to the `map` primitive on the new value received and sends it further.

The listener is then added to the finalizers list so that this when the initial event dies, the

```scala
def map[B](f: T => B): Event[B] = {
    val out: EventSink[B] = new EventSink[B]()
    val l: Listener = listen(out.node, (trans: Transaction, a: T) => {
        out.send(trans, f.apply(a))
    })
    out.addCleanup(l)
}
```

Listing 14: Implementation of the map primitive in Rhein

ones depending on it die as well.

Chapter 4

UI Binding Library

The UI Binding Library is a separate component of Rhein . Actually, we can think of it as a sub-library. The FRP engine in Rhein is totally independent of this sub library. As we mentioned at the beginning of Chapter 3, its purpose is to provide a quick way to display reactive values on the web and to provide the FRP system means to interact with the external environment.

4.1 The framework pipeline

The UI Binding library relies heavily on ScalaJS and Scalatags. The Framework pipeline is presented in figure 4.1.

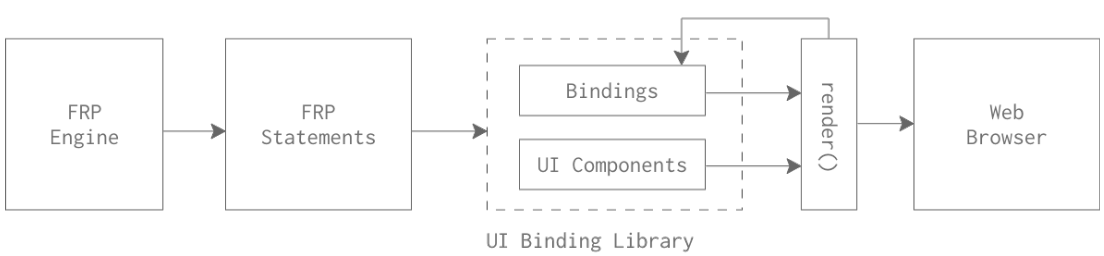


Figure 4.1: Framework pipeline.

FRP statements are created by the developer and once the application logic is done, the developer can inject these values in UI components that are later added to the DOM. At this point, if the compiler intercepts `Behaviour` values in the context of the render process (values that haven't been wrapped in the provided UI elements) it will use the implicit conversions provided by the library and injects them in proper wrappers that can be rendered.

4.1.1 ScalaJS & Scalatags

ScalaJS

ScalaJS is a library that provides a way to build robust front-end web applications. It is actually a Scala to JavaScript compiler that focuses on three objectives:

- Correctness. ScalaJS provides strong typing for JavaScript code (which is untyped by default). ScalaJS spares developers from silly mistakes like mixing up strings or numbers, forgetting what key an object has or worrying about typos in method names.
- ScalaJS optimizes Scala code to highly efficient JavaScript. It uses an incremental compilation mechanism that guarantees speedy turn-around times when code changes.
- Interoperability is the last objective. In ScalaJS you can use JavaScript libraries including React and AngularJS.

In our project, ScalaJS is used to compile our FRP engine together with our UI Binding library to JavaScript that makes it easier to create web applications that work on FRP logic. At the same time, ScalaJS provides a quick way to create a mechanism where reactive variables that are displayed on the web are updated accordingly making sure they always display the correct value.

Scalatags

Scalatags is a small, fast HTML & CSS construction library for Scala. The library transforms fragments that look like this:

```
div(cls := "main")(h1("Hello"))
```

into HTML code like this

```
<div class="main"><h1>Hello</h1></div>
```

Scalatags is hosted on Maven Central and it is very simple to install in a project. We simply add the following line to our build.sbt file.

```
libraryDependencies += "com.lihaoyi" %% "scalatags" % "0.8.2"
```

The library works seamlessly with ScalaJS and the combination of the two provides Rhein with a quick way of displaying values on the web. Please find below, a small application that leverages the combined functionality of these two libraries.

```scala
// imports here
@JSExportTopLevel("Main")
object Main {
    def main(args: Array[String]) {
        dom.document.body.appendChild(
            div(cls := "main")(
                h1("Hello world")
            ).render
        )
    }
}

// to run the app, compile the to js code using
// "fastOptJS" in your sbt, and import it into an html file.
```

Listing 15: Small application with ScalaJS and Scalatags

4.2 Bindings

The UI Binding library provides some special implicit functions that wrap Behaviour objects into DOM elements that are forced to re-render when the value of the internal behaviour changes. This mechanism leverages the implicit functions in Scala.

There are three implicit functions in the UI Binding Library: `Behaviour` to `Modifier`, `Behaviour` to `AttrValue` and `Behaviour` to `StyleValue`. We will explain them in order below:

- The first implicit function converts a Behaviour value to a Modifier object. The Modifier type is a trait that all HTML tags provided in Scalatags inherit. Therefore it can be used to store any HTML tag, and it can be rendered as well. Inside this function, we attach a listener to the `changes()`(which returns a stream of events with updates on the value of the behaviour) method and forcing the wrapper element to re-render with the new value. Therefore whenever we include a behaviour inside Scalatags code, the rendered value will be always up to date, and it will change automatically.

- The second implicit function is used to convert a Behaviour value that is used as an attribute value in an HTML tag in Scalatags. For example, in the code

  `val width = new Behaviour(150);`

  `renderToDOM(img(..., width := width))`

  the image tag will have a width equal to the value of the behaviour's current value. If the behaviour will change its value (in this case it won't because it's a constant behaviour) the image will be re-rendered with the corresponding size.

- The final implicit function acts very similarly to the previous one, but it converts Behaviours to `StyleValue` instead of `AttrValue`. `StyleValue` is the type used to store CSS values for HTML tags in Scalatags. Therefore, if we use a Behaviour value as a style value, it will re-render the tag with the new updated value for the style (e.g. `renderToDOM(div(..., backgroundColor := color))`, where `color` is a Behaviour).

These functions make it extremely easy and quick to display reactive values on the web. The limitation to using these is that you can't make use of the event streams that we provide in Rhein . There is no way of converting DOM events into stream events using implicit functions. To solve this, we provide a few UI elements that work with both behaviours and event streams which can take advantage of DOM listeners and therefore create stream of events that interact with the external environment.

4.3 UI Components

The UI Binding library provides a few UI elements that we can use to display values on the web, but also to emit values from click events or retrieve input from the users and use these in our FRP logic.

4.3.1 Button

The `Button` component corresponds to a `div` element that is attached with an `onClick` DOM listener. Moreover, the Button component is injected with an event stream called `eventClicked` which emits a `Unit` value whenever we click on it. The `onClick` DOM listener is where this `eventClicked` triggers the `send()` method. This property is publicly available on the `Button` class, and therefore we can use it to create FRP logic in our program. To create a button, you use

```
val button: Button = new Button("Click me!")
```

and to render it on the DOM, you pass the `button.domElement` to the render or HTML fragment in your code.

4.3.2 Label

The `Label` component corresponds to a simple `p` element that wraps the value of a behaviour and re-renders whenever the value of behaviour changes. When creating a `Label` component we must provide a Behaviour in the constructor:

```scala
val label: Label = new Label(myBehaviour)
```

If you prefer to pass the behaviour object directly to the render or the HTML fragment in your code, you can do so – the implicit functions that we mentioned in the previous section will take care of the conversion. Use of `Label` is optional.

4.3.3 TextField

The `TextField` component corresponds to an `input(type = "text")` element. In the implementation of this component we can find both behaviours and streams of events. There are two event streams that are merged into one, and one behaviour that holds the contents of the text field (the behaviour is created using the merged event and the `hold` primitive, therefore the value changes whenever either of the events fire).

The first event, can be provided by the developer if the value inside the text field needs to be changed. Therefore, if we emit a value on this stream of events, the value inside the text field will reflect the value emitted.

The second event stream is used internally to update the value of the behaviour (which holds the current text in the text field) when the user types a new string inside the field. This is done by attaching an `onInput()` DOM listener to the `text` tag which emits the value associated with the DOM event inside the stream event.

To create a simple `TextField` we use

```scala
val textField: TextField = new TextField("Initial value")
```

4.3.4 Listing

`Listing` is a special component created specifically for situations where you want to map a collection (stored inside a behaviour) to DOM elements, and force re-render the collection whenever its contents changes. It is simple to display a single value to the web, either using `Label` or using the behaviour object itself (which automatically gets converted), but when it comes to collections, we need a way to describe how each element will be rendered. Therefore, when using the `Listing` component, we must provide a function where we define how each element in the collection will be rendered in the browser.

To better illustrate the use case of this component, we will look at a small example. Assume you work on a blog application where you display articles. Each article has a list of comments. The comments list contains objects of type `Comment` which has two properties: `content` and date. In this example, we omit to display the article and just deal with displaying the list of comments on the web. This list is stored inside `val comments: Behaviour[List[Comment]]`. To display the comments, we will use the `Listing` component and we will provide a function that will be applied to each comment inside the list that specifies how we want them to be rendered.

```
class Comment(var content: String, var date: String)
// response is a list of comments that we fetched from
// a server or something similar
val comments: Behaviour[List[Comment]] = new Behaviour(response)

val listing: Listing = new Listing(comments, (comment: Comment) => {
    div( cls := "comment")(
        p(comment.content),
        span(cls := "date")(comment.date)
    )
})

//renderToDOM(listing.domElement)
```

Listing 16: Example of how to use Listing

Chapter 5

Implementation

In this chapter, we discuss the prior FRP systems implementations and the way they influenced us while writing Rhein 's implementation.

5.1 FRP Implementations

FRP systems usually use either a push-based or a pull-based implementation.

Pull-based systems work by sampling Behaviours over a sequence of discrete times. This evaluation model tends to be more appropriate for processing continuous behaviours because they change often. In contrast to the pull-based (also known as demand-driven) in the push-based model, evaluation is performed at every event occurrence instead of repeatedly sampling values. Because of this, behaviours and events are constructed in a way that results in building a dependency graph. Every time an event fires, the payload associated with the firing is pushed into the dependency graph and behaviours are then recomputed as the new values traverse through the graph to produce output.

Research has shown that simple push-based implementations suffer from wasteful recomputations of behaviours [15]. In Rhein we followed the same principle and idea that Sodium library is using, that is, evaluation is made based on the order of the rank nodes which partially avoids the wasteful recomputations. Research provides a bunch of further improvement techniques and optimisations, but these were omitted in Rhein in order to try to keep the implementation as simple and minimal as possible.

5.2 First attempts

The first attempts of implementing Rhein were based on different inspiration sources than the ones mentioned in Chapter 3 where we provide the structure and conceptual model of the framework. The first point of reference was Odersky's online course *Functional Programming Principles in Scala* [28] that is available on *Coursera*[1]. The course also provides a section on FRP where Odersky is presenting a minimal implementation of an FRP system that is based on Scala.React (that is the implementation of his paper "Deprecating the Observer Pattern"[5]). The provided FRP system from his course only focuses on behaviours (which are called Signals in this implementation) while discrete events were left out.

The Signals (or as we referred to them in this paper, behaviours) in his implementation are reactive values that can depend on other Signals, therefore they can be used to create dependency graphs corresponding to relationships in a program. While the dependency tracking in our FRP system relies on the observer pattern, in Odersky's system, the dependencies are created using *Dynamic Variables* (which are similar to *ThreadLocal* in Java, and provide a non-intrusive way to store and pass around context/thread-specific information). As a result of using this idea (and thanks to the functional features of Scala as well) dependency relations can be expressed in an innovative way. Listing 17 provides an example of how to create relations using this system.

```scala
val a = new Var(1)
val b = new Var(2)
val sum: Signal[Int] = a() + b() // sum depends on a and b
```

Listing 17: Creating dependencies using Odersky's Signal type

`Var` is an extension of the `Signal` class that allows mutation on the value held by the signal. Therefore, in Listing 17 the variable `sum` depends on `a` and `b`, and whenever `a` or `b` changes, `sum` is automatically updated.

Unfortunately, as the goal of this project is to provide a UI Binding Library alongside the FRP system so that we can easily create web applications out of the box, this idea turned out to be less suitable for our purposes. The binding mechanism of this implementation style was not very efficient, especially from the development experience point of view. Use of intensive DOM listeners from ScalaJS were required to make it possible to display a Signal on the web.

The implementation style in Sodium was the one that would get our project closer our goal the most, and therefore a decision has been made to follow this one.

[1] https://www.coursera.org/learn/progfun1

5.3 Internal Components in Rhein

Rhein uses the observer pattern under the hood. This is a common choice in push-based FRP systems. In a push-based system, if there is an event or behaviour that has a new value then it pushes that value to its dependent nodes. This model is often referred to as a data-driven because propagation is driven by when an event happens instead of on demand (the latter refers to the pull-based model).

5.3.1 Ranking

As we mentioned in section 3.3 that explains the FRP life cycle, the first stage of the program consists of building a directed acyclic graph (DAG) that represents the dependencies in our application. The order in which dependencies are executed is determined using a ranking system which is inspired by the Sodium library. Each `Event` has a class property named `node` with the type `Node`. The `Node` class is a helper type that lies at the heart of the ranking system and helps the framework decide which nodes/events execute first. This is done in two steps:

1. First, we assign ranks (a number of type `Long`) to each event object based on walking the directed graph and ensuring that invariants hold (i.e., dealing with loops – which are covered in section 3.11)

2. Secondly, we put every outstanding job into a `PriorityQueue` and we pull them off the head executing them in rank order.

This idea and the sorting algorithm are initially inspired from Elliott and Hudak paper "Deprecating the Observer Pattern". These ideas are implemented in the Sodium library as well.

Each time we create an `Even` or `Behaviour` or we apply a transformation on them using the 7 primitives that Rhein provides (which are covered in the Chapter 3), under the hood, dependency relations are created by linking their corresponding `Nodes`.

The important section where nodes are linked and indirectly creating dependencies is presented in Listing 18. The `listen()` method available in the `Event` class is where nodes are linked to each other using the `node.linkTo()` method that is part of the `Node` class. In this function, we also add the action that will be executed when this `Event` receives a new value, in the list with the other listeners of this `Event`. Moreover, this function returns a new listener implementation object that provides the functionality to unlisten this newly attached listener (i.e. remove listeners/stop listening).

```
def listen(
    target: Node,
    trans: Transaction,
    action: TransactionHandler[T]
): Listener = {
    node.linkTo(target)
    listeners += action
    new ListenerImplementation[T](this, action, target)
}
```

Listing 18: Implementation of listen function in Rhein

5.3.2 Transactions

In some FRP systems, a transaction is called a "moment" and this is a better name because it is conceptual and not operational. However, we decided to call them "transactions" because they reflect the same behaviour that we find in database systems contexts.

In Rhein if you do not create a transaction, one is created automatically. For instance, each `send()` method from the `EventSink` class runs in a new transaction (see line 3 and 4 in Listing 19). The concept of transactions uses the *Loan Pattern* [29], where you pass in a function to the method that creates a transaction representing the code you want it to execute in the transactional context. In Rhein you can create a transaction explicitly using the `Transaction.run` method (see line 7 in code snipped 19).

```
1 // Sending events creates transactions automatically
2 val eventSink: EventSink[Int] = new EventSink()
3 eventSink.send(1)       // --> Transaction 1
4 eventSink.send(2)       // --> Transaction 2
5
6 // Creating an explicit transaction
7 Transaction.run((transaction: Transaction) => {
8     // execute code inside transaction
9 })
```

Listing 19: Illustrating how transactions are created in Rhein

The *Loan Pattern* is a software design pattern in which a function that manages some resource is given a fragment of code in the form of a function to run, and the function loans the resource to the passed code. The reason for this is to reduce the chance of resources to leak making it impossible for the caller to forget to close the resource. A common example of this pattern is reading from a file. When you read from a file, opening and closing the file is done on your behalf and you just provide the code that is accessing the resource (i.e. reading lines in that file).

The `Transaction` class is relatively simple and only has three methods. Inside a transaction, there are two class properties: `pq` that has a type of `PriorityQueue[Entry]` and `last` of type `List[Runable]`. Two of the methods we mentioned above are just helper functions to add new elements in the two collections. The third function is called `close()` and it is responsible with running the outstanding jobs (updates that are a result of processing an incoming event) that we stored in the Priority Queue, and later, running the actions we stored the `last` property. The class `Entry` is just a helper class that acts like a pair that holds two values: a node and an action. It also provides a way to compare Entries based on the rank number, which helps to maintain the correct order in the Priority Queue. The `Transaction` class comes along with a companion object that contains the two main methods which facilitate running code in the context of a transaction. Listing 20 provides you with the two methods. The first method `evaluate()` is the one that is used inside the FRP engine in Rhein . The second methods is not used internally and it is provided for the developer's own usage.

```
        // Used internally while creating Events and
        // Behaviours or applying primitives
        def evaluate[A](code: Transaction => A): A = {
            val trans: Transaction = new Transaction()
            try {
                code(trans)
            } finally {
                trans.close()
            }
         }
        // Additional method to run a piece of code
        // inside a transactional context
        def run(code: Handler[Transaction]) {
            val trans: Transaction = new Transaction()
            try {
                code.run(trans)
            } finally {
                trans.close()
            }
         }
```

Listing 20: Implementation of the Transaction Companion Object

Rhein similarly to Sodium, executes transactions in two steps:

1. Process all fired events in an `Event` stream simultaneously

2. Update all behaviour values atomically

During step 1, behaviours cannot change their values/state, therefore event processing sees a single ”moment” representing the state before the transaction started. All events processed in a single transaction can be viewed as truly simultaneous to each other.

In step 2, you then apply all the updates that have been queued in step 1, atomically. Atomically means it is impossible to observe a situation where some updates have been applied and not others.

5.4 Implementing Events

The `Event` type corresponds to the event abstraction from the FRP literature. The `Event` class holds a value that is available at discrete points in time and acts like a stream or pipe where values travel from one side to another; it also provides different operations that can be used to create dependencies and to create more complex structures that have diverse behaviours.

5.4.1 Properties

The `Event` type defines a set of core properties as follows:

The `listeners` property is a mutable list (`List` in Scala is immutable, while `ListBuffer` is a mutable version of a List) of `TransactionHandlers` (handler that runs in a context of a Transaction) that stores all the actions of all listeners attached to this Event. When an event is fired, we loop through this list and run all actions in the corresponding transaction.

The `finalizers` property is the place where we store all listeners resulted from calling the internal method `listen()` from this event. Whenever we listen to an event we return a `ListenerImplementation` object that extends the `Listener` trait and provides a `unlisten` method that can be used to stop listening to this event. All these listener objects are stored so that when this event gets killed, we loop through the listeners attached an unlisten all of them. The loop runs in the `finalize()` method.

The `finalize()` method is a callback that the JVM will run when it is looking for finalizeable objects, therefore instead of garbage collecting the object it, schedules to be collected in the next round. This helps with memory management.

The `node` property is part of the ranking system that we described in Section 5.3.1, and helps with creating the dependency graph in the initialisation phase.

```scala
class Event[T]() {
    trait Listener {
      def unlisten()
    }

    trait TransactionHandler[A] {
      def run(trans: Transaction, a: A)
    }

    // list with listeners on this event
    protected var listeners = new ListBuffer[TransactionHandler[T]]()

    // List with all listeners that need to be removed when this
    // event gets killed
    protected var finalizers = new ListBuffer[Listener]()

    // Each Event has a Node object that is used to create the graph
    var node: Node = new Node(0L);

    // More code ...
}
```

Listing 21: Core class properties of Event

5.4.2 Methods

Apart from the class properties, the `Event` class provides a number of methods that either build the functionality of the event mechanism itself or add the possibility to perform operations such as map and filter to the event stream. The latter operations are part of the 7 primitives that the framework provides.

One of the most important methods in the class is `listen()`. We described its mechanism in Section 5.3.1, but we will describe it in more detail here. In fact, there are three methods for listen but they have different parameters. Two are used internally to link dependencies and one is provided for the developer's own usage. Nonetheless, the listen method that the developer calls on events, actually calls the other listen methods. Together, they form a chain so that when you listen to an event like here,

```scala
val l: Listener = event.listen(payload => println(payload))
```

the call chain looks like the one described in Figure 5.1.

Step 1, corresponds to the listen method in the line of code above and has as a parameter of type `Handler` which can be seen as a function that is passed along. Therefore the action in

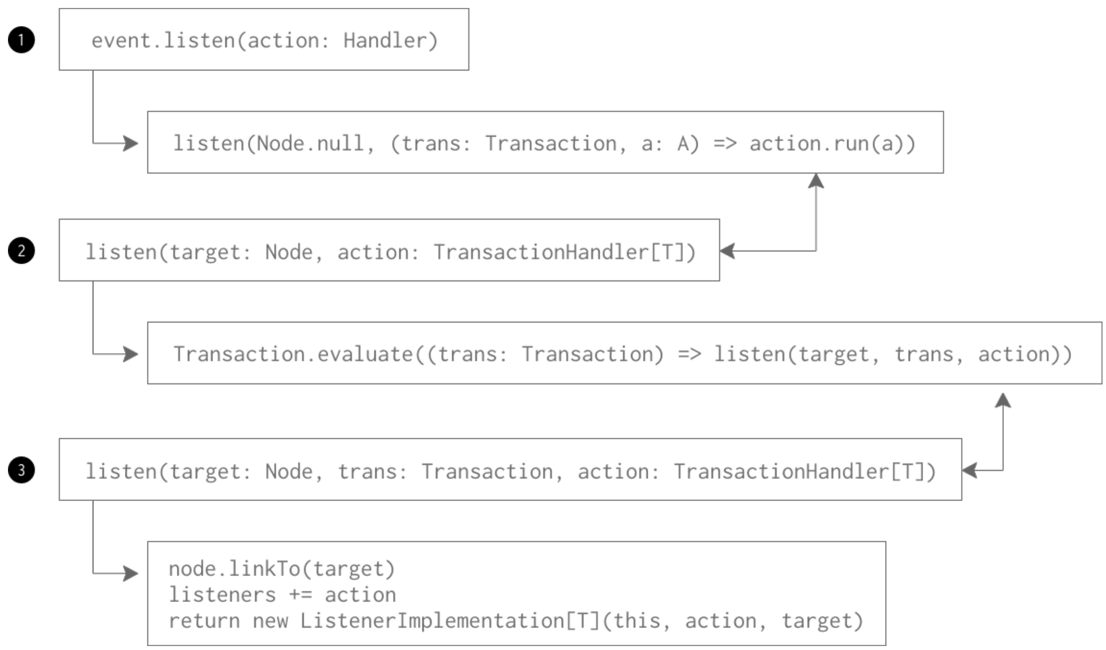


Figure 5.1: Conceptual view of the chain call of listen.

this case is just printing the received value. The first step, calls the second `listen` variation and transforms the handler into a transactional handler because the second `listen` takes as a parameter a `TransactionHandler` instead of `Handler`. On top of that, we also provide a target node that we link to the event that we listen to. In this example, we'll just pass a Null node object, because we're not creating an actual dependency in this case. The node parameter is important when we implement some of the primitives (i.e., map, filter etc.). In the second step, the second `listen()` function then calls the third `listen()` variation in a transactional context, passing the newly created transaction down the chain as well. In the last step, the third `listen()` variation links the target node to the node of this event, adds the action to the list of listeners and returns a `Listener` object.

The primitive functions (i.e., `map`, `hold`, `filter` and `snapshot`) are used to return instances of `Event` objects (except for the `hold` which returns a `Behaviour` object). The primary purpose for each primitive is to initialize new `Event` objects that emit values with certain mechanisms (e.g. `map` transforms one stream of type `A` to type `B`) that are usually determined by functions that are provided by the developer, whenever the initial event fires (the initial event emits values and the second one emits transformed values – this depends on the type of primitive we use). Implicitly, the primitive operations also add dependency connections between inputs (initial event), transformation functions and the event (or behaviour while using `hold`) that is being returned. The way each primitive works is presented in Chapter 3. Their implementations differ accordingly. For example, the implementation of the filter primitive gives a good blueprint on how the other primitives are constructed as well:

```
1 def filter(f: T => Boolean): Event[T] = {
2     val out = new EventSink[T]()
3     val l = listen(out.node, (trans: Transaction, a: T) => {
4         if (f(a)) out.send(trans, a)
5     })
6     out.addCleanup(l)
7 }
```

Listing 22: Implementation of filter primitive

In the implementation provided in Listing 22, on line 2 we create a new `EventSink` (which is an extension of `Event` class that provides an additional method called `send()` that explicitly emits events). On line 3 we create a `Listener` on the current event where we pass the node of the newly created event and a transaction handler in which we call the `send()` method on the `out` event with the transaction and received value. Since the filter primitive is responsible with transforming a stream of events to one that only accepts certain values (which are determined by the function provided as a parameter `f`) we add a condition and check the value `a` respects the filter condition before emitting it to the result event. The implementation ends with returning the new event, on which we add the `l` listener to its `finalizers` list (this is done using the `addCleanup()` function).

5.4.3 Companion Object

In addition to the `Event` class, Rhein provides a companion object that exposes the `merge` primitive that is responsible with merging two streams of events into a single one and a method called `interval` that creates a special event stream that emits a `Unit` value at an interval provided as a parameter. This latter function is useful when we need to schedule a specific event to occur at certain times repeatedly.

5.5 Implementing Behaviours

The `Behaviour` class corresponds to the behaviour abstraction (also named signal) that we find in the FRP literature. A `Behaviour` is a container of a value that changes over time, but that always has a value. This value can be sampled at any time. Similar to the `Event` type, behaviours also provide the possibility to apply operations on them in order to create more complex application logic.

The implementation of `Behaviour` class is much smaller compared to the `Event`. This is because `Behaviour` relies on the `Event` type internally, and therefore the main part of its mechanism is actually the `Event` itself. At the same time, there are only two primitives we can apply to behaviours: `map` and `lift`

5.5.1 Properties

The `Behaviour` class contains one `Event` stream internally. This stream of events corresponds to the internal changes in the value that the `Behaviour` holds. When we create a new behaviour we can either explicitly provide an event stream or one is automatically created for us. Besides the event stream (that is optional) we must provide an initial value. This is because behaviours always have a value – we can think of them as memory cells. This can be done using either of the constructors available, which we included in Listing 23.

```
// Primary constructor
class Behaviour[T](var event: Event[T], var value: Option[T]) {
    // Auxiliary constructor
    def this(value: Option[T]) {
        this(new Event[T](), value)
    }
    // More code...
}
```

Listing 23: Constructors of Behaviour class

At the time of creating a behaviour, in a transactional context, we start listening to the event stream of the class and update the value as we receive new events. Inside this listener, we check if there's a `valueUpdate` available, and if not – we make sure the update of the value of this behaviour is done at the end of the current transaction and before any transaction after it. The purpose of this is that all event streams during the same transaction can snapshot the same value of any one behaviour (consistency among snapshots). The code is presented in Listing 24.

5.5.2 Methods

The `Behaviour` class provides several methods that we can use to sample the current value, combine behaviours (i.e., `lift`) or map them similarly to event streams. Their signatures can be found in Listing 25.

```
Transaction.evaluate((trans1: Transaction) => {
    this.cleanup = Some(
        event.listen(Node.NullNode, trans1, (trans2: Transaction, a: T) => {
            if (Behaviour.this.valueUpdate.isEmpty) {
                trans2.last(new Runnable() {
                    def run() {
                        Behaviour.this.value = valueUpdate
                        Behaviour.this.valueUpdate = None
                    }
                })
            }
            this.value = Some(a)
        })
    )
})
```

Listing 24: Implementation of the Behaviour Class. This runs whenever we create a new Behaviour.

```
// sample the current value
def sampleNoTrans(): T = value.get

// exposes the internal event
def changes(): Event[T]

// map primitive
final def map[B](f: T => B): Behaviour[B]

// lift primitive
final def lift[B, C](b: Behaviour[B], f: (T, B) => C): Behaviour[C]
```

Listing 25: Signatures of the methods of Behaviour Class

Chapter 6

Applications

The examples provided throughout this paper, introduce to the reader how abstractions and concepts work, but they do not fully explain the usefulness and practicality of implementing applications using Rhein . In this chapter, we present two substantial examples, which demonstrate the efficiency and scalability of using Rhein to implementation interactive web applications.

6.1 Todo Application

The first application that we will provide in this chapter, is a todo web application where the user can create a list of tasks and mark them as done as he pleases. The application is relatively simple, but it demonstrates how you can use Rhein to develop such application logic and gives a good blueprint of how you should structure your code while developing using this library. These examples are just a suggestion, and they do not represent the only way of doing things in Rhein . They were created with the purpose to make it easy to convert your operational thought process into a conceptual one.

6.1.1 Application structure

The application is structured in two phases. Phase 1 represents the logic of the application, and we use the FRP engine inside Rhein to write the main functionalities of the program. The second phase contains the UI and the rendering process. These phases were separated to make it easy to provide this as a boilerplate if you want to start writing your own application in Rhein .

Phase 1 - Application Logic

In this application, you can enter a new todo task using a text field. The tasks will be displayed below each other and you will be able to mark them as complete. After marking a task as done, it will disappear.

To model the tasks, we created a `Task` class that holds properties such as the task description and a flag that tells us if the task is done. Moreover, each task object contains a unique id that is generated at the time of creating a new task, using `java.util.UUID.randomUUID`. This is important because we need a way to identify each task in our list so that we can apply updates later. Please refer to Listing 26 for the code of the Task class.

```
class Task(var description: String, var isDone: Boolean) {
    val id: String = java.util.UUID.randomUUID.toString
}
```

Listing 26: Task class in our todo application

There is a small detail that was omitted while describing the `Button` from the UI Binding Library. As we mentioned before, the component is injected with an event stream that emits events whenever the button is clicked. The type of the event stream inside `Button` is `Message` which is just a simple trait. The reason for this design choice is to facilitate attaching particular event streams that we want to fire with a specific `Message` when the button is clicked. In this example, we use this idea to implement the functionality of marking a task done (deleting it).

If we think about the application in terms of relations, there is a stream of events that changes the state of our application. Inside this stream of events, we emit a special value `TodoMessage` which describes the type of operation we need to make on our list of tasks. The object that is being fired contains an `action` which in this example can either be `"add"` or `"remove"` and a `value`. The purpose of the second property is to be able to identify which task we add or remove. We created a special class that implements the `Message` trait available in the UI Binding Library, listed below.

```
class TodoMessage[T](var action: String, var value: T) extends Message
```

The code that creates the stream of events can be found in Listing 27. The code starts with initialising two UI components which are needed because they are two of the source of events in our program. On line 6 we start implementing the logic of our application. The event stream called `todoEvents` is a stream on which we fire messages of type `TodoMessage` that describe the action we want to apply to our state. For now, the event fired in `todoEvents` is an action of type "add" associated with a new task that contains the text inserted in the text field at the time of clicking the add button (the value inside the text field is captured using the snapshot primitive).

```
        // UI elements
        val buttonAdd = new Button("+")
        val taskTextField = new TextField("Insert a task here...")

        // Stream of event messages
        val todoEvents = buttonAdd.eventClicked
            .snapshot(taskTextField.text, (click, text: String) => {
                new TodoMessage[Task]("add", new Task(text, false))
        })
```

Listing 27: Stream of event messages

Now that we have set up the communication with the external environment (the user actions in the browser) we must create the state of our application. We have mentioned before in this paper that Behaviour acts like a memory cell. Therefore, when we need to store a time-varying value (in this example, the list of tasks) in Rhein , Behaviour is the best choice. Since there is an accumulation of events that all have to capture the current version of the state and add new information to it, we need to implement an accumulator (similar to the accumulator we provided in Section 3.11 about Loops and Accumulators).

```
        val state = new BehaviourLoop[List[Task]]
        val updates = todoEvents.snapshot(state,
            (event, _state: List[Task]) => {
                event.action match {
                    case "add" => {
                        event.value :: _state
                    }
                    case "remove" => {
                        _state.filter(p => p.id != event.value.id)
                    }
                }
            }
        )
        state.loop(updates.hold(List()))
```

Listing 28: Accumulator code for our todo application

The code provided above implements an accumulator where the state is being updated according to the action received on the todoEvents stream. Inside the snapshot, we apply a different operation on the list depending on the action of the value emitted.

Phase 2 - UI and Rendering

The application logic is roughly done. There is still one small detail that we need to implement but we'll discuss that here.

In order to display our list of tasks, we will use the `Listing` component provided in the UI Binding library. The component is designed for behaviours that hold collections inside them. In our application we have a `Behaviour[List[Task]]` and `Listing` seems like a perfect choice. Inside the list component we provide a function that specifies how each task will be rendered. The code is available in the listing below.

```scala
// Helper function to create the HTML structure of a task
// The class names of the html tags, are part of Bootstrap (CSS framework)
def todoItem(task: Task, buttonDelete: Button) = {
    div(cls := "d-flex align-items-center")(
        div(cls := "d-flex align-items-center border p-2")(
            span(cls := "mr-2", task.description),
            if (!task.isDone)
                span(cls := "badge badge-pill badge-warning", "Ongoing")
            else
                span(cls := "badge badge-pill badge-success", "Done")
        ),
      buttonDelete.domElement(cls := "btn btn-light ml-2")
    )
}

// List component
val list = new Listing[Task](cState, (task: Task, index: Int) => {
    val buttonDelete = new Button("-")
    buttonDelete
        .attachEvent(todoEvents, new TodoMessage[Task]("remove", task))
    todoItem(task, buttonDelete)
})
```

Listing 29: Creating a list of tasks ready to be rendered

Inside the listing, we create new buttons for each task in order to complete the application logic. These buttons help implementing the delete functionality. This is done by attaching the `todoEvents` to these buttons and provide a new message to emit, which, in this case is a `TodoMessage` corresponding to deleting a task. Therefore, whenever we click on a button, the application knows which particular event to delete. This events will travel through the stream and the state will change accordingly.

To complete the implementation, all we have to do is add the UI components to the DOM and render them. Please find the final part of the implementation below.

```
        dom.document.body.innerHTML = ""
        dom.document.body.appendChild(
            div(cls := "mt-3")(
                h1("Todo Application"),
                br,
                div(cls := "d-flex align-items-center")(
                  buttonAdd.domElement,
                  taskTextField.domElement
                ),
                br,
                br,
                list.domElement
            ).render
        )
```

Listing 30: Rendering the UI

6.1.2 Conclusion

The complete code can be found in Appendix A. In conclusion, in this application, we present how we can combine different functionalities using a single event stream. This application can be implemented further very easily. For example, if we want to add an `"edit"` action, all we have to do is add a new edit button and display the description of the task inside a text field instead of just rendering it directly to the DOM. Then, using `snapshot` we can capture the new edited value inside the text field and pass it to the event stream, and eventually update the state.

6.2 Game of Life

The Game of Life is a cellular automaton created by the British mathematician John Horton Conway in 1970 [30]. The game is a 0-player game, meaning that its evolution is determined by its initial state, without requiring further input. The user interacts with the Game of Life by creating an initial configuration and observing how it evolves.

6.2.1 Rules

Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The rules have been sourced from the official Game Of Life Wiki page [30].

6.2.2 Implementing the Rules

The game rules have been implemented in such a way that all functions are reverentially transparent. This makes it easy to chain these operations and eventually only call a final function with a state to generate a new state. In Listing 31 we present the signature of the methods that implement the game logic. These are quite standard because Game of Life has a well defined set of rules. These methods do not use any elements from Rhein , and we consider that their implementation is not essential for the purpose of this example, therefore we only provide the signatures here. They are implemented entirely in Scala. The full implementation of all methods can be found in Appendix A.

6.2.3 Game Loop

Writing this kind of simulation using Rhein is much simpler that one might think. The application can be separated into two parts: the game rules which we already described above, and the game loop.

To model the grid in the application, we created a `World` type that corresponds to a `ListBuffer[Boolean]`. We use a single array to store the rectangular map, therefore, the length of the list will be equal to the `WIDTH * HEIGHT`, which are the dimensions of the grid. Each element inside the list corresponds to a cell which can be false or true (dead or alive).

The logic of the game loop is simple. We have an event stream that emits an occurrence every 500 milliseconds (this can be changed to a different interval if needed). Every time we process this event (in other words, at each game tick), we use `snapshot` to get the current state of the world (which is wrapped inside a behaviour), apply the `updateWorld` method which

```
// Converts a 2D matrix to a World type
def createWorld(initial: ListBuffer[ListBuffer[Int]])

// Returns true if cell at (x, y) is alive, false otherwise
def isAlive(x: Int, y: Int, world: World, width: Int, height: Int)

// Returns number of alive cells around cell (x, y)
def getNumberOfNeighbours(x: Int, y: Int, world: World,
    width: Int,height: Int
): Int

// update a cell at (x, y) according to the rules of the game
def updateCellState(x: Int, y: Int, world: World,
    width: Int, height: Int
): Boolean

// apply update to all cells
def updateWorld(world: World, width: Int, height: Int): World
```

Listing 31: Implementing game rules for Game of Life

generates the new state according to the rules, and feeds it back inside the behaviour using the `hold` primitive. An accumulator is created here as well, as in the previous application because the new state of the behaviour is created using the previous state and additional operations. All these details can be seen in Listing 32.

```
type World = ListBuffer[Boolean]
val INTERVAL = 500L
// event that emits every 500 milliseconds
var tickStream: Event[Unit] = Event.interval(INTERVAL, INTERVAL)

// Create an empty state using one of the provided examples
var eventLoop: EventLoop[List[World]] = new EventLoop()
var state = eventLoop.hold(List(createWorld(pattern2)))

// Accumulator
eventLoop.loop(tickStream.snapshot(state,
    (event, _state: List[World]) => {
        List(updateWorld(_state.head, WIDTH, HEIGHT))
    })
)
```

Listing 32: Game loop of Game of Life application

The World has been wrapped inside a List so that we can use the `Listing` object from the

UI Binding Library to render the map later. As a consequence of this decision, we could easily display multiple grids on the web at the same time.

6.2.4 Pausing the Game

The Game of Life is a simulation that runs over time, therefore we thought that a pause functionality would be useful for the user. To implement the pausing feature, we create a new behaviour describing if the game should be paused or not. The source of events that changes the state of the pause is the pause button. To be able to implement a toggle (pause and resume) we use an accumulator. The new state will be equal to the negation of the previous state (this makes sense because we're using a `Boolean` to model this). Please find the implementation below.

```
        var buttonPause: Button = new Button("Pause")
        var pauseLoop: EventLoop[Boolean] = new EventLoop()

        // Holds the state paused or not
        var pauseState = pauseLoop.hold(true)
        pauseLoop.loop(
          buttonPause.eventClicked
            .map(click => true)
            .snapshot(pauseState, (event, _pauseState: Boolean) => {
                !_pauseState
            })
        )

        // Filterin out tick events when we are in pause state
        tickStream = tickStream.filter(x => pauseState.sampleNoTrans())
```

Listing 33: Implementing the pause feature in Game of Life

After creating the pause state, all we have to do is filter out any game ticks while `pauseState` is true, therefore forcing the game to not update which results in a pause. To resume the game, the user must click on the pause button again.

6.2.5 Rendering the World

To render the world we use the `Listing` component. We provide a function that maps over the list of worlds (in this example, there is just one) and transforms each World into an HTML table. Please refer to Listing 34 for the code.

```scala
// Creating the listing component
var grid: Listing[World] =
    new Listing(cState, (world: World, index: Int) => {
        makeGrid(world)
    })

// returns a single <td> element that will be a black
// box if the cell is alive, white box otherwise
def cellTd(alive: Boolean) = {
    var color = if (alive) "#000" else "#fff"
    td(style := s"width: 15px; height: 15px;
        border: 0.5px solid #ccc; background-color: ${color};")
}

// Looping through the world and create the <table>
def makeGrid(world: World) = {
    table(style := "border-collapse: collapse")(
        tbody(world
            .sliding(HEIGHT, HEIGHT)
            .toList
            .map((row: ListBuffer[Boolean]) => {
                tr(row.map((cell: Boolean) => {
                    cellTd(cell)
                }))
            })
        )
    )
}

// Appending UI elements to the DOM
dom.document.body.innerHTML = ""
dom.document.body.appendChild(
    div(cls := "mt-3")(
        h1("Game of Life"),
        grid.domElement,
        buttonPause.domElement
    ).render
)
```

Listing 34: Rendering the world in the Game of Life application

6.2.6 Conclusion

The complete code can be found in Appendix A. In conclusion, in this application, we present how we can implement a simple game loop using an accumulator and a special event `interval` that emits events at a given rate. This application can be implemented further very easily. For example, we can render multiple simulations at the same time. This can be done by adding more worlds in the state of the application, and adjust the accumulator to update all words instead of just one.

Chapter 7

Legal & Professional Issues

During the implementation of the project, great concern has been shown to abide by the *Code of Conduct* which is issued by the *British Computer Society (BCS)* [31]. Great care has been taken in this project to make sure any Open-Source code or libraries used are explicitly stated. This project consists of: my own work; Open-Source libraries; and Open-Source code provided on the internet.

In this chapter we will discuss certain aspects that we enforced while developing Rhein and that will also serve as a guideline for further development of this library. The end goal of the project is to create a library which aims to improve the development process of interactive applications on the web. Moreover, the end product will be open-sourced and therefore, open-source guidelines have been followed to ensure professional standards and legal compliance.

According to Github, every open source project should include the following documentation: Open source license, README, Contributing guidelines, Code of conduct [32].

Open source license

An open-source license guarantees that other parties can use, copy, modify, and contribute back to a project without repercussions. It also protects the author from legal situations. A licence is required when open sourcing a project.

Rhein is designed to be used as a dependency in other projects. Moreover, there are project dependencies inside Rhein as well: Scalatags and ScalaJS. Both these dependencies have a permissive (common permissive licenses include MIT, Apache 2.0, ISC, and BSD. licence) and that means we can use a permissive licence as well (there are no licensing restrictions imposed by the dependencies used inside Rhein). According to the Open Source Guide provided by

GitHub [32] MIT licence is the most common among projects that are meant to be used as a dependency. MIT licence is a short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications and larger works may be distributed under different terms and without source code [33]. In conclusion, Rhein will have an MIT license.

README

The source code of Rhein is hosted on GitHub. Following the guidelines for open-source project, Rhein provides a README file describing the goals of the project, how to use it. Other information regarding the project like handling contributions and information about licenses and attribution are also present.

Contributing guidelines

A contributing file tells a project's audience how to participate in the project. For example, this might include information on: how to file a bug report (try using issue and pull request templates) or how to suggest a new feature. Rhein provides a contributing guideline in the root of the project directory.

Code of conduct

A code of conduct helps set ground rules for behaviour for the project's participants. This is especially valuable if launching an open source project for a community. A code of conduct empowers to facilitate healthy and constructive community behaviour.

Note

As we mentioned before, the guidelines mentioned above are to be used while developing further versions of Rhein and to ensure professional standards. There have been no direct contributions to this software and the software consists of my own work except where explicitly stated so.

Chapter 8

Results/Evaluation

In this chapter, we will present certain aspects of the project and showcase the performance and limitations of Rhein .

8.1 Results

8.1.1 The declarative nature

We showed how in Rhein dependency relations are created automatically, and we can easily convert relationship diagrams representing features into code written in Rhein . This aspect empowers the developer to think in a declarative way and therefore ensures working in the "problem space" rather than the "machine space". Using FRP, the sequence of execution is derived from the relationships in the program and this gives developers the opportunity to focus more on the *what* rather than the *how*.

8.1.2 Interoperability

Rhein turns out to be a quick way to integrate FRP code in existing applications. Rhein provides simple functions to interpolate with operational code, and since it has been developed in Scala, it can be compiled to other languages as well. The UI Binding library inside Rhein , can be translated to other targets as well. For example, we could easily implement a different UI Binding library targeting Desktop applications using ScalaFX [1].

[1] UI DSL written within the Scala Language that sits on top of JavaFX

8.1.3 Unit Testing

Since writing code in Rhein will result in code that is referentially transparent where functions do not have side-effects and implicit state does not exists, makes it easy to write unit tests. This is a big advantage in the development process and ensures that the quality of code is high.

8.1.4 Replacing callbacks

In many cases, Event streams in Rhein can be considered as a drop-in replacement for listeners, callback or observer pattern mechanism. By preventing the use of observer pattern, we automatically decrease the bugs in an application. In Chapter 2 we listed 6 common bugs that are caused by the observer pattern.

8.1.5 Less is more

FRP implementations tend to be less lengthy compared to ones using other paradigms. As a consequence, refactoring code becomes an easier task and the chances of hitting the complexity wall decrease. Eventually, the complexity of programs written in FRP will be smaller. This is mainly because dependency tracking is handled under the hood, and developers only have to focus on explicitly defining behaviours in their application.

8.2 Evaluation and Limitations

The purpose of this research project is to illustrate the benefits of FRP in writing event-driven applications with a specific target for the web. The current framework cannot be used in production mode, under any circumstance, and it is advised not to. Intense testing and optimisations are required to be able to successfully publish this library on a package manager so that it can be used by the developer community. The current version of Rhein should constitute as a proof of concept only. As a consequence of this, testing during development has not been of high importance. On the other hand, we provide a few performance tests measuring the differences between using FRP and using traditional event-handling while implementing Game of Life (please refer to Section 6.2 for how it works and implementation in Rhein)

During the development process, and also while reflecting back on this project, we identified several limitations that the current version of the framework holds.

- The current version of Rhein , mainly focuses on discrete events only. A true FRP system covers both discrete and continuous time. An important primitive usually called *switch* in the FRP literature that brings continuity to an FRP system, has not been implemented. The reason for this is the limited resources available on this paradigm. A lot of research papers provide the implementation of this primitive in ambiguous languages like Scheme and Haskell.

- Rhein lacks proper memory management and optimisations on its internal mechanics. This was omitted on purpose to make the implementation as simple and as easy to understand.

- Going back to the true definition of FRP, Maier et al. stated that a true FRP system must have denotational semantics. The 7 primitives available in Rhein are based on the ones provided in the Sodium library, which provides denotational semantics[2] for its implementation. The primitives inside Rhein have the same semantics as the ones provided by Sodium library, but their implementation has been modifier while integrating them inside Rhein . We believe that in order for Rhein to be considered a true FRP system, denotational semantics should be provided for this implementation as well.

8.3 Game of Life FRP vs Non-FRP

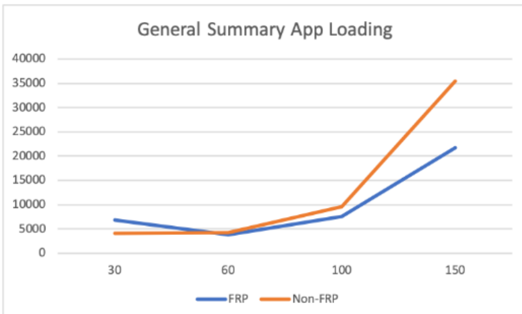
In this section, we will present the results of our tests on two different implementations of Game of Life. The first implementation was done using Rhein and the second one, using event listeners (the traditional observer pattern). The implementation of the FRP Game of Life can be found in Section 6.2. The code for the non-FRP implementation can be found in Appendix A. Both implementations use the same functions for the game logic. The game loop is the one that has been implemented according to the paradigms. The tests have been performed inside Google Chrome (Version 80.0.3987.163 (Official Build) (64-bit)) Dev Tools, on the Performance tab.

The machine we ran our tests is an Apple MacBook Pro (early 2015) with a 2.7 GHz Dual-Core Intel Core i5 processor, 8GB of memory that is running Mac OS Catalina 10.15 (beta). For the first two benchmarks, execution time is given in milliseconds and is the arithmetic mean of 10 individual runs. Additionally, we included one benchmark presenting how much memory does the program consume while running, and one illustrating the frames per second of the simulation. All tests have been run on 4 different map sizes: 30x30, 60x60, 100x100 and 150x150.

[2] Denotational semantics for Sodium can be found on the link below
https://github.com/SodiumFRP/sodium/tree/master/denotational

8.3.1 Application Loading

The following graphs show how much time does the application need to load entirely. This process involves scripting, which is the initialisation of the program, rendering that represents time spend computing styles associated with each DOM node and their location, and painting which is the actual time spent drawing pixels on the browser. The second graph illustrates the time spend on scripting alone. Rendering and painting do not have a meaning in our tests, because the rendering mechanism is the same for both implementations.



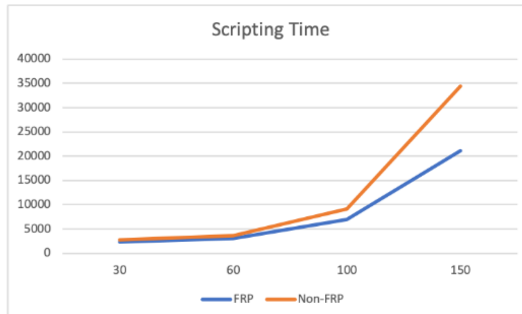


Figure 8.1: Time taken for the application to load on the left. Out of the total time taken, the graph on the right shows how much time was taken for scripting. The total time usually is equal to scripting + painting + rendering.

The results show a slightly better performance for the FRP implementation. We can also notice that the time is increasing at a rapid pace, especially between the two last world sizes. We attempted to run tests on higher map dimensions but they were unsuccessful due to the crashing of the application.

8.3.2 Memory & FPS

The first graph on the left represents how much memory does the application take to run. This memory represents the HEAP memory that JavaScript objects and DOM elements are using. On the right side, we provide a graph that shows how the FPS of the simulations decreases.

The results show that the memory of both implementations are similar, and moreover they tend to increase at the same pace. On the other hand, the frames per second while running the simulation are quite low. The UI/UX standards for animations/simulations should not go below 30FPS, while 60FPS is highly recommended.

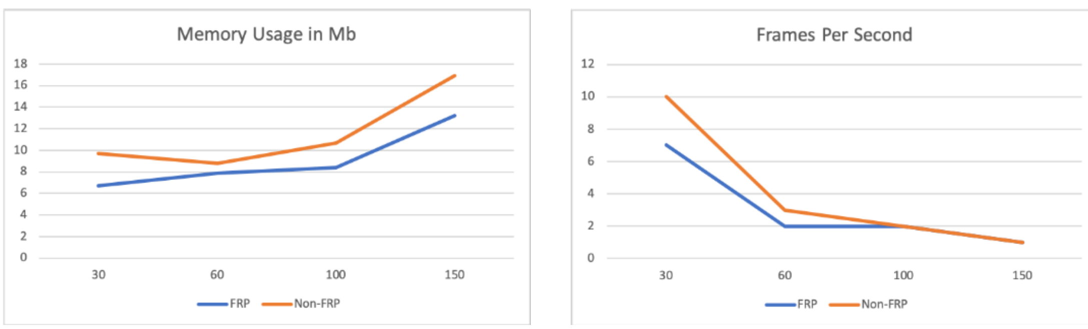


Figure 8.2: The amount of memory consumed by the application is presented on the left graph. FRP of the simulation is presented in the right.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In Chapter 2 we presented one of the biggest issues that developers face while implementing event-driven applications, the complexity wall. According to Blackheath and Jones, software quality and the costs of accomplishing it is a serious issue for the development industry, and we totally agree with this. Software is being used to solve more and more complex problems. Therefore, we need stronger techniques to deal with the greater complexity, and that's why the industry is looking at what functional programming can offer. FRP is part of that and we think for a certain type of projects, especially ones that involve a lot of event handling, FRP can help turn intractable code into maintainable code so you can navigate the complexity barrier.

Rhein aims to provide developers with a small library based on FRP abstractions to explore the benefits of FRP and further grow the community revolving around these ideas. The goal of this is to give enough proof and resources to understand and make it easy to explore this new paradigm.

The framework provides two main data types that reflect the two abstractions from the FRP literature: Events and Behaviours. Events are a way of building pipelines or streams of data, where you can apply several operations (or as we called them in this project, primitives) to create complex behaviours and relations in your application. Behaviours model the state in the application, therefore you can use them to store any data that your application needs while running. Behaviours work hand in hand with stream of events, and we provided a few primitives that make the conversion between the two, in order to provide more functionalities. Behaviours always have a value, and you can sample them at any time. We also introduced the concept of transactions, which we used to implement a small scheduling system where events fired in the same transactions are considered simultaneous and updates on behaviours happen in an atomic way. This ensures that the application is always in a valid state and therefore gives developers the opportunity to focus on what the application needs to do and no how it should do it. Lastly, we illustrated ways Rhein can be used to implement certain functionalities that your application might need. Moreover, we provided two applications developed entirely in Rhein to showcase the benefits and ways it can be used.

9.2 Further Improvements

As we mentioned in the Evaluation section of this report, the current version of Rhein is far from a production-ready framework. This section outlines further avenues of research to get the library closer to a production-ready version and to better improve the benefits Rhein brings to developing interactive web applications.

Denotational Semantics and Continuous Time

As we mentioned in the Evaluation chapter, in order for this library to be considered a true FRP system, we must provide denotational semantics and further improve the continuous abstractions inside Rhein . This can be accomplished with further research.

Syntax Improvements

The syntax of some of the primitives inside Rhein can be further improved. For example, the `lift` primitive could be used like this `val c: Behaviour[Int] <- a + b` instead of `val c: Behaviour[Int] = a.lift(b, (a_, b_) => a_ + b_)`. Another example of syntax improvements is infix operators for primitives. For example, we could write `((i + 1) <@ clickEvent)` for the `snapshot` primitive instead of this `clickEvent.snapshot(i, (click, i_) => i_ + 1)`. This can be accomplished using implicit functions that the Scala language provides.

Visualisation and Debugging Tools

Visualisation and debugging tools could be an idea of providing a graphical representation of FRP state over time. This would allow the developer to debug FRP logic without analysing the FRP engine implementation. Another idea which the Elm [34] language already has, would be a time-travelling debugger where we can go back and forth in time of the application state.

9.2.1 Visual Programming

Blackheath and Jones have suggested an interesting idea of visual programming using FRP in their book [1]. Since FRP code can be easily translated from relationship diagrams we could create programs using a visual interface.

References

[1] S. Blackheath and A. Jones. *Functional Reactive Programming*. Manning Publications, 2016. ISBN 9781633430105. URL https://books.google.co.uk/books?id=a0OzrgEACAAJ.

[2] Wikipedia contributors. Spaghetti code — Wikipedia, the free encyclopedia, 2019. URL https://en.wikipedia.org/w/index.php?title=Spaghetti_code&oldid=928736080. [Online; accessed 10-February-2020].

[3] Javier Pérez and Yania Crespo. Computation of refactoring plans from refactoring strategies using htn planning. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, page 24–31, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315005. doi: 10.1145/2328876.2328880. URL https://doi.org/10.1145/2328876.2328880.

[4] Benjamin Erb. Concurrent programming for scalable web architectures. Diploma thesis, Institute of Distributed Systems, Ulm University, April 2012. URL http://www.benjamin-erb.de/thesis.

[5] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the observer pattern. Technical report, 2010.

[6] Bob Reynders, Dominique Devriese, and Frank Piessens. Multi-tier functional reactive programming for the web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 55–68, 2014.

[7] Riccardo R Pucella. Reactive programming in standard ml. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No. 98CB36225)*, pages 48–57. IEEE, 1998.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 0201633612.

[9] Jonathan Edwards. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 925–932, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587684. doi: 10.1145/1639950.1640058. URL https://doi.org/10.1145/1639950.1640058.

[10] Side effect (computer science) - wikipedia. https://en.wikipedia.org/wiki/Side_effect_(computer_science). (Accessed on 04/17/2020).

[11] Gregory H Cooper. Integrating dataflow evaluation into a practical higher-order call-by-value language. 2008.

[12] Parent. Sean. A possible future of software development, 2008. URL https://stlab.cc/legacy/figures/Boostcon_possible_future.pdf.

[13] Wikipedia. Functional reactive programming, 2019. URL https://en.wikipedia.org/wiki/Functional_reactive_programming#Formulations_of_FRP.

[14] Conal Elliott and Paul. Hudak. Functional reactive animation. *ICFP*, 1997. URL http://conal.net/papers/icfp97/.

[15] Lamont Samuels. Declarative computer graphics using functional reactive programming. page 200. URL http://knowledge.uchicago.edu/record/585.

[16] Wikipedia. Principle of compositionality - wikipedia. https://en.wikipedia.org/wiki/Principle_of_compositionality, . (Accessed on 02/27/2020).

[17] Bonér Jonas, Farley Dave, Kuhn Roland, and Thompson Martin. The reactive manifesto. https://www.reactivemanifesto.org/, April 2014. (Accessed on 02/29/2020).

[18] Google. Angularjs — superheroic javascript mvw framework. https://angularjs.org/, 2010. (Accessed on 02/29/2020).

[19] Facebook. React – a javascript library for building user interfaces. https://reactjs.org/, 2013. (Accessed on 02/29/2020).

[20] Apple Inc. Wwdc19 - apple developer. https://developer.apple.com/wwdc19/, June 2019. (Accessed on 02/29/2020).

[21] Reactive Cocoa. Reactivecocoa/reactiveswift: Streams of values over time. https://github.com/ReactiveCocoa/ReactiveSwift. (Accessed on 02/29/2020).

[22] Reactivex/rxswift: Reactive programming in swift. https://github.com/ReactiveX/RxSwift. (Accessed on 02/29/2020).

[23] ReactiveX. Reactivex - intro. http://reactivex.io/intro.html. (Accessed on 02/29/2020).

[24] Conal Elliott. Specification for a functional reactive programming language - stack overflow. https://stackoverflow.com/questions/5875929/specification-for-a-functional-reactive-programming-language. (Accessed on 02/29/2020).

[25] Julian M Hedges. *Towards compositional game theory*. PhD thesis, 2016.

[26] Stephen Blackheath. Sodiumfrp/sodium: Sodium - functional reactive programming (frp) library for multiple languages. https://github.com/SodiumFRP/sodium. (Accessed on 03/21/2020).

[27] Li Haoyi. lihaoyi/scala.rx: An experimental library for functional reactive programming in scala. https://github.com/lihaoyi/scala.rx. (Accessed on 03/21/2020).

[28] Martin Odersky. Functional programming principles in scala — coursera. https://www.coursera.org/learn/progfun1. (Accessed on 04/08/2020).

[29] Josh Suereth. *Scala In Depth*. 2011. URL http://www.manning.com/suereth/. This is not complete, only available in ”Early Access” edition.

[30] Wikipedia. Conway’s game of life - wikipedia. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life, . (Accessed on 04/11/2020).

[31] BCS. bcs-code-of-conduct.pdf. https://cdn.bcs.org/bcs-org-media/2211/bcs-code-of-conduct.pdf. (Accessed on 04/12/2020).

[32] Github. Starting an open source project — open source guides. https://opensource.guide/starting-a-project/#launching-your-own-open-source-project. (Accessed on 04/12/2020).

[33] Mit license — choose a license. https://choosealicense.com/licenses/mit/. (Accessed on 04/12/2020).

[34] Evan Czaplicki. Elm - a delightful language for reliable webapps. https://elm-lang.org/. (Accessed on 04/14/2020).

Appendix A

Source Code

A.1 FRP Engine

A.1.1 Event

```scala
package rhein

import scala.collection.mutable.ListBuffer
// Listener trait
// must implement a unlisten method

/**
  * Listener trait
  */
trait Listener {
  def unlisten()
}

// TransactionHandler[A]
// must implement a run method

/**
  * Transaction Handler is used to implement listeners
  */
trait TransactionHandler[A] {
  def run(trans: Transaction, a: A)
}

// Also known as Stream
// Stream of events that fire at discrete times

```

```
/**
  * Event, also known as Stream in other FRP systems,
  * is a stream of events that fire at discrete points
  * in time
  *
  */
class Event[T]() {
  // List with listeners on this event
  protected var listeners = new ListBuffer[TransactionHandler[T]]()
  // Collects all listeners that need to be removed when this event gets killed
  protected var finalizers = new ListBuffer[Listener]()
  // Uset to identify each event
  var node: Node = new Node(0L);

  protected val firings = ListBuffer[T]()

  /**
    * Listener implementation. When creating a new listener on an event,
    * it returns an instance of this class and then the
    * listener can be closed/killed
    *
    * @param event
    * @param action
    * @param target
    */
  final class ListenerImplementation[T](
      event: Event[T],
      action: TransactionHandler[T],
      target: Node
  ) extends Listener {

    /**
      * Unlisten method that breaks the dependency
      * relation
      */
    def unlisten() = {
      event.listeners -= action
      event.node.unlinkTo(target)
    }

    override protected def finalize() = {
      unlisten()
    }
  }

  /**
```

```scala
   * Listen for firings of this event. The returned Listener has an unlisten()
   * method to cause the listener to be removed. This is the observer pattern.
   * @param action
   * @return
   */
  def listen(action: Handler[T]): Listener = {
    listen(Node.NullNode, (trans: Transaction, a: T) => { action.run(a) })
  }

  /**
   * Listeners used for implementing primitives
   * @param target
   * @param action
   * @return
   */
  def listen(target: Node, action: TransactionHandler[T]): Listener = {
    Transaction.evaluate((trans: Transaction) => listen(target, trans, action))
  }

  /**
   * The final listener method of the whole listen chain
   * that creates dependencies and adds actions to the
   * list of actions of this event
   *
   *
   * @param target
   * @param trans
   * @param action
   * @return listener implementation that can be used to unlisten
   */
  def listen(
      target: Node,
      trans: Transaction,
      action: TransactionHandler[T]
  ): Listener = {
    node.linkTo(target)
    listeners += action
    new ListenerImplementation[T](this, action, target)
  }

  /**
   * Map Primitive
   *
   * @param f transformation function
   * @return
   */
```

```scala
  def map[B](f: T => B): Event[B] = {
    val out: EventSink[B] = new EventSink[B]();
    val l: Listener = listen(out.node, (trans: Transaction, a: T) => {
      out.send(trans, f.apply(a));
    })
    out.addCleanup(l)
  }

  /**
    * Hold primitive
    *
    * @param initValue required as behaviours always have a value
    * @return
    */
  final def hold(initValue: T): Behaviour[T] = {
    Transaction.evaluate(trans => new Behaviour[T](this, Some(initValue)))
  }

  /**
    * Filter primitive
    *
    * @param f filtering function
    * @return
    */
  def filter(f: T => Boolean): Event[T] = {
    val out = new EventSink[T]()
    val l = listen(out.node, (trans: Transaction, a: T) => {
      if (f(a)) out.send(trans, a)
    })
    out.addCleanup(l)
  }

  /**
    * Snapshot primitive
    *
    * @param b behaviour being snapshoted
    * @param f combination function
    * @return
    */
  def snapshot[B, C](b: Behaviour[B], f: (T, B) => C): Event[C] = {
    val out = new EventSink[C]()
    val l: Listener = listen(out.node, new TransactionHandler[T]() {
      def run(trans: Transaction, a: T) {
        out.send(trans, f(a, b.sampleNoTrans()))
      }
    })
```

```
    out.addCleanup(l)
  }

  /**
    * Helper that adds listeners to
    * the list of finalizers of this eventg
    *
    * @param l
    * @return
    */
  def addCleanup(l: Listener): Event[T] = {
    finalizers += l
    this
  }

  /**
    * Removes the listeners while this event is killed
    */
  override def finalize() {
    finalizers.foreach(_.unlisten)
  }
}

/**
  * Companion Object
  */
object Event {

  /**
    * Merge primitive
    *
    * @param ea
    * @param eb
    * @return
    */
  def merge[T](ea: Event[T], eb: Event[T]): Event[T] = {
    val out: EventSink[T] = new EventSink[T]()
    val h = new TransactionHandler[T]() {
      def run(trans: Transaction, a: T) {
        out.send(trans, a)
      }
    }

    val l1 = ea.listen(out.node, h)
    val l2 = eb.listen(
```

```
      out.node,
      new TransactionHandler[T]() {
        def run(trans1: Transaction, a: T) {
          trans1.prioritized(out.node, new Handler[Transaction]() {
            def run(trans2: Transaction) {
              out.send(trans2, a)
            }
          })
        }
      }
    )
    out.addCleanup(l1).addCleanup(l2)
  }

  /**
    * Special event interval, that emits at a given rate
    *
    * @param delay
    * @param period
    * @return
    */
  def interval(delay: Long, period: Long): Event[Unit] = {
    val out: EventSink[Unit] = new EventSink()
    val t = new java.util.Timer()

    val task = new java.util.TimerTask {
      def run() = out.send(Unit)
    }
    t.schedule(task, delay, period)

    out
  }
}
```

A.1.2 EventSink

```
package rhein
import scala.collection.mutable.ListBuffer

class EventSink[T] extends Event[T] {

  /**
    * After creating the dependencies,
    * loop with this function.
    * @param a
    */
```

```
  def send(a: T) {
    Transaction.evaluate((trans: Transaction) => { send(trans, a) })
  }

  /**
    * Get all listener actions and send
    * this payload down the pipe
    *
    * @param trans
    * @param a
    */
  def send(trans: Transaction, a: T) {
    try {
      this.listeners
        .clone()
        .asInstanceOf[ListBuffer[TransactionHandler[T]]]
        .foreach(_.run(trans, a))
    } catch {
      case t: Throwable => t.printStackTrace()
    }
  }

}
```

A.1.3 EventLoop

```
package rhein

/**
  * Event Loop provides a way to create circular dependencies
  * and also helps in creating accumulators
  *
  */
class EventLoop[T] extends EventWithSend[T] {

  private var ea_out: Option[Event[T]] = None

  def loop(initStream: Event[T]) {
    if (ea_out.isDefined)
      throw new RuntimeException("StreamLoop looped more than once")
    ea_out = Some(initStream)
    addCleanup(initStream.listen(this.node, new TransactionHandler[T]() {
      override def run(trans: Transaction, a: T) {
        EventLoop.this.send(trans, a)
      }
    }))
```

```
  }
}
```

A.1.4 EventWithSend

```
package rhein

/**
  * Modified version of send used to implement
  * event loops.
  */
class EventWithSend[T] extends Event[T] {

  /**
    * Send method used to trigger events
    * associated with a pyload
    * inside a transaction
    *
    * @param trans
    * @param a
    */
  def send(trans: Transaction, a: T) {
    if (firings.isEmpty)
      trans.last(new Runnable() {
        def run() { firings.clear() }
      })
    firings += a

    try {
      listeners.clone.foreach(_.run(trans, a))
    } catch {
      case t: Throwable => t.printStackTrace()
    }
  }

}
```

A.1.5 Behaviour

```
package rhein

/**
  * Behaviour - time varying value
  * Differentiates from an Event by always having a value => continuous
  * @param event
  * @param value
```

```
  */
class Behaviour[T](var event: Event[T], var value: Option[T]) {
  var valueUpdate: Option[T] = None
  var cleanup: Option[Listener] = None

  // Auxiliary constructor
  def this(value: Option[T]) {
    this(new Event[T](), value)
  }

  // Creates the behaviour dependency
  // Using a Transaction
  // Listens for changes on the injected event stream
  Transaction.evaluate((trans1: Transaction) => {
    this.cleanup = Some(
      event.listen(
        Node.NullNode,
        trans1,
        (trans2: Transaction, a: T) => {
          // make sure the updates happen at the end of a transaction
          // and before any new one
          if (Behaviour.this.valueUpdate.isEmpty) {
            trans2.last(new Runnable() {
              def run() {
                Behaviour.this.value = valueUpdate
                Behaviour.this.valueUpdate = None
              }
            })
          }
          this.valueUpdate = Some(a)
        }
      )
    )
  })

  /**
    * Sample the current value without a transaction
    *
    * @return the current value of the behaviour
    */
  def sampleNoTrans(): T = value.get

  /**
    * Returns the newst value of the behaviour
    *
    * @return
```

```
     */
  def newValue(): T = {
    valueUpdate.getOrElse(sampleNoTrans)
  }

  /**
    * Returns an event with all changes that have been made to this
    * Behaviour. Gives you the discrete updates to a behaviour (effectively
↪   the inverse of hold())
    * @return
    */
  def changes(): Event[T] = {
    event
  }

  /**
    * Map primitive
    *
    * @param f transformation function
    * @return
    */
  final def map[B](f: T => B): Behaviour[B] = {
    changes().map(f).hold(f.apply(sampleNoTrans()))
  }

  /**
    * Lift primitive
    *
    * @param b to be combined with
    * @param f combination function
    * @return
    */
  final def lift[B, C](b: Behaviour[B], f: (T, B) => C): Behaviour[C] = {
    def ffa(aa: T)(bb: B) = f(aa, bb)
    Behaviour.apply(map(ffa), b)
  }

  override def finalize() = {
    cleanup.get.unlisten
  }

}

/**
  * Companion Object
  */
```

```
object Behaviour {

  /**
    * Helper constructor for the lift primitive
    * Combines two behaviours together
    *
    * @param bf
    * @param ba
    * @return
    */
  def apply[T, B](bf: Behaviour[T => B], ba: Behaviour[T]): Behaviour[B] = {
    val out = new EventSink[B]()
    var fired = false
    def h(trans: Transaction) {
      if (!fired) {
        fired = true
        trans.prioritized(out.node, { trans2 =>
          out.send(trans2, bf.newValue().apply(ba.newValue()))
          fired = false
        })
      }
    }
    val l1 = bf
      .changes()
      .listen(out.node, new TransactionHandler[T => B]() {
        def run(trans: Transaction, f: T => B) {
          h(trans)
        }
      })
    val l2 = ba
      .changes()
      .listen(out.node, new TransactionHandler[T]() {
        def run(trans: Transaction, a: T) {
          h(trans)
        }
      })
    out
      .addCleanup(l1)
      .addCleanup(l2)
      .hold(bf.sampleNoTrans().apply(ba.sampleNoTrans()))
  }
}
```

A.1.6 BehaviourSink

```
package rhein

/**
  * Force an update on the behaviour.
  *
  * @param initValue
  */
class BehaviourSink[T](initValue: Option[T])
    extends Behaviour[T](new EventSink[T](), initValue) {

  def send(a: T) {
    event.asInstanceOf[EventSink[T]].send(a)
  }
}
```

A.1.7 BehaviourLoop

```
package rhein

/**
  * Behaviour Loop provides a way to create circular dependencies
  * and also helps in creating accumulators
  *
  */
final class BehaviourLoop[T] extends Behaviour[T](new EventLoop[T](), None) {

  /**
    * After creating the dependencies,
    * loop with this function.
    *
    * @param a_out
    */
  def loop(a_out: Behaviour[T]) {
    event match {
      case s: EventLoop[T] => s.loop(a_out.changes())
      case _               =>
    }
    value = Some(a_out.sampleNoTrans)
  }

  override def sampleNoTrans(): T = {
    if (value.isEmpty)
      throw new RuntimeException("CellLoop sampled before it was looped")
    value.get
  }
```

```
}
```

A.1.8 Transaction

```
package rhein

import java.lang.Comparable
import collection.mutable.PriorityQueue
import scala.collection.mutable.ArrayBuffer
import java.util.concurrent.atomic.AtomicLong

/**
  * Simple handler trait
  * that must implement a run method
  */
trait Handler[T] {
  def run(a: T)
}

/**
  * Helper class to create the priority queue
  *
  * REF! - Based on
  * https://github.com/SodiumFRP/sodium/blob/master
  * /scala/src/main/scala/sodium/Transaction.scala
  *
  * @param rank
  * @param action
  */
class Entry(var rank: Node, var action: Handler[Transaction])
    extends Comparable[Entry] {
  override def compareTo(other: Entry): Int = {
    rank.compareTo(other.rank)
  }
}
class Transaction() {
  private val pq = new PriorityQueue[Entry]()
  private var last: List[Runnable] = List()

  /**
    * Add a new transaction that is prioritized
    * and runs before everything
    *
    * @param rank
    * @param action
    */
```

```scala
  def prioritized(rank: Node, action: Handler[Transaction]) {
    pq += new Entry(rank, action)
  }

  /**
    * Add a new action that is NOT prioritized
    * and runs last
    *
    * @param action
    */
  def last(action: Runnable) {
    last = last ++ List(action)
  }

  /**
    * Close the transaction
    * Run all actions in pq and last
    * in this specific order
    */
  def close() {
    while (!pq.isEmpty)
      pq.dequeue().action.run(this);

    last.foreach(_.run())
    last = List()
  }
}

object Transaction {

  /**
    * Method to facilitate running the
    * specified code inside a single transaction, with the contained
    * code returning a value of the parameter type A.
    *
    * @param code code to be executed inside the transactional context
    * @return value of the returned code function
    */
  def evaluate[A](code: Transaction => A): A = {
    val trans: Transaction = new Transaction()
    try {
      code(trans)
    } finally {
      trans.close()
    }
  }
```

```scala

  /**
    * Method to facilitate running a piece of code
    * inside a transactional context
    * @param code code to be executed inside the transactional context
    */
  def run(code: Handler[Transaction]) {
    val trans: Transaction = new Transaction()
    try {
      code.run(trans)
    } finally {
      trans.close()
    }
  }
}
```

A.1.9 Node

```scala
package rhein

import scala.collection.mutable.HashSet

/**
  * Node class that is used to implement
  * the ranking system
  *
  * Used to make sure events are executed
  * in proper order
  *
  * REF! - Based on:
  * https://github.com/SodiumFRP/sodium/blob/master/scala/src/main/scala/sodium/Node.scala
  *
  * @param rank
  */
class Node(var rank: Long) extends Comparable[Node] {
  import Node._
  val listeners: HashSet[Node] = HashSet()

  /**
    * Links one node to another
    * Creates a new connection in the dependecy
    * graph
    *
    * @param target
    * @return
    */
```

```scala
  def linkTo(target: Node): Boolean =
    if (target == NullNode) {
      false
    } else {
      val changed = target.ensureBiggerThan(rank, Set())
      listeners.add(target)
      changed
    }

  /**
    * Breakes a connection between two nodes
    * Removes a dependency
    *
    * @param target
    */
  def unlinkTo(target: Node) {
    if (target != NullNode)
      listeners.remove(target)
  }

  /**
    * Ensures that the nodes added inside
    * the dependency graph are ordered by the ranks
    *
    * @param limit
    * @param visited
    * @return
    */
  private def ensureBiggerThan(limit: Long, visited: Set[Node]): Boolean = {
    if (rank > limit || visited.contains(this)) {
      false
    } else {
      val accVisited = Set(this) ++ visited
      rank = limit + 1
      listeners.forall(_.ensureBiggerThan(rank, visited))
    }
  }

  /**
    * Helper method to compare two
    * nodes by rank
    *
    * @param o
    * @return
    */
  override def compareTo(o: Node): Int =
```

```
    if (rank < o.rank) -1
    else if (rank > o.rank) 1
    else 0
}

/**
  * Companion Object
  */
object Node {
  // Null node, used when implementing the
  // listen method for developer
  val NullNode = new Node(Long.MaxValue)
}
```

A.2 UI Binding Library

A.2.1 Label

```
package rhein.ui
import rhein._
import scala.scalajs.js
import scalatags.JsDom.all._

/**
  * A simple UI Label component that was injected with an Behaviour
  * To facilitate interoperability
  *
  * @param text
  */
class Label(text: Behaviour[String]) {

  val initialValue = text.sampleNoTrans()
  // UI - using Scalatags
  val element = p(style := "margin: 0")(initialValue)
  var domElement = element.render

  // Logic
  var listener = text
    .changes()
    .listen(x => {
      val newLast = p(style := "margin: 0")(x).render
      domElement.parentElement.replaceChild(newLast, domElement)
      domElement = newLast
    })
}
```

A.2.2 Button

```
package rhein.ui
import rhein._
import scala.scalajs.js
import scalatags.JsDom.all._


/**
  * Simple class representing the initial value emited
  * by the click event
  */
class EmptyMessage extends Message


/**
  * A simple UI Button component that was injected with an Event Stream
  * To facilitate interoperability
  *
  * @param text
  * @param label used to provide an css ID
  */

class Button(
    val text: String,
    val label: String,
) {

  // Injection
  var valueToEmit: Message = new EmptyMessage()
  var eventClicked: Event[Message] = new Event()
  var eventClickedSink: EventSink[Message] = new EventSink()
  eventClicked = eventClickedSink

  // UI - using Scalatags
  val domElement = div(id := label, cls := "btn btn-primary", onclick := { () =>
    {
      eventClickedSink.send(valueToEmit)
    }
  })(text)

  /**
    * helper method to inject a new event stream
    * inside this component and provide a specific
    * value to emit
    *
    * @param event
```

```scala
    * @param newValueToEmit
    */
  def attachEvent(event: Event[_], newValueToEmit: Message) {
    valueToEmit = newValueToEmit
    eventClickedSink = event.asInstanceOf[EventSink[Message]]
  }

  /**
    * Auxiliary constructor
    *
    * @param text
    * @return
    */
  def this(text: String) {
    this(text, "")
  }

}
```

A.2.3 TextField

```scala
package rhein.ui
import rhein._
import scala.scalajs.js
import scalatags.JsDom.all._
import org.scalajs.dom.{Event => DomEvent}

/**
  * Text field component corresponding to a
  * <input type="text"/> HTML component that has an
  * event injected
  *
  * @param sText
  * @param initialValue
  */
class TextField(
    var sText: Event[String],
    var initialValue: String,
) {

  // Injection
  final val userChangesSink: EventSink[String] = new EventSink()
  var userChanges: Event[String] = new Event()

  userChanges = userChangesSink
  val merged = Event.merge(userChangesSink, sText)
```

```
  var text: Behaviour[String] = merged.hold(initialValue)

  sText.listen((newVal) => {
    domElement.value = newVal
  })

  // <input> element using scalatags
  val element =
    input(`type` := "text",
      cls := "form-control",
      value := text.sampleNoTrans,
      style := "max-width: 250px")

    // property used to render this component to the dom
  var domElement = element.render

  /**
    * attaching a DOM listener where we emit events
    * whenever the text inside the input changes
    */
  domElement.oninput = (event: DomEvent) => {
    val newVal = domElement.value.toString
    userChangesSink.send(newVal)
  }

  /**
    * Auxiliary constructor
    *
    * @param initialValue
    * @return
    */
  def this(initialValue: String) {
    this(new Event[String], initialValue)
  }
}
```

A.2.4 Listing

```
package rhein.ui
import rhein._
import scala.scalajs.js
import scalatags.JsDom.all._
import scalatags.JsDom.TypedTag
import scala.collection.{Iterable => Iter}

/**import
```

```scala
  * A simple UI Listing component that was can be injected
  * with a behaviour that holds a list of items and a function describing
  * how to render each element in the list
  * @param value
  * @param f maps an element from the list to a dom element
  */

class Listing[T](
    value: Behaviour[List[T]],
    f: (T, Int) => scalatags.JsDom.Modifier
) {

  // Injection
  val initialValue: List[T] = List()
  val element = span(
    for ((elem, index) <- initialValue.toSeq.zipWithIndex) yield f(elem, index)
  )

  var domElement = element.render

  // Force re-render every time a new value is received
  var listener = value
    .changes()
    .listen((newVal: List[T]) => {
      val newLast =
        span(
          for ((elem, index) <- newVal.toSeq.zipWithIndex) yield f(elem, index)
        ).render
      domElement.parentElement.replaceChild(newLast, domElement)
      domElement = newLast
    })
}
```

A.2.5 Message

```scala
package rhein.ui

// Simple trait that represents the generic type of values that
// can be transmited through the UI Binding library
trait Message
```

A.2.6 Bindings

```scala
package rhein.ui
import org.scalajs.dom.html
import org.scalajs.dom.{Element}
```

```scala
import scala.scalajs.js
import scalatags.JsDom.all._

import rhein._

/**
  * Bindings is just a wrapper
  * for implicit functions that convert
  * behaviours into dom elements
  *
  * REF! - Ideas based on:
  * https://www.youtube.com/watch?v=i9mPUU1gu_8
  */
object Bindings {

  /**
    * Converts Behaviour that is used in the
    * context of a dom element in scalatags to
    * an actual dom element (Modifier)
    *
    * @return Modifier
    */
  implicit def BehaviourToDom[T](r: Behaviour[T])(
      implicit f: T => Modifier
  ): Modifier = {
    var initialValue = r.sampleNoTrans()

    // UI - using Scalatags
    val element = span(initialValue)
    var domElement = element.render

    // Logic
    var listener = r
      .changes()
      .listen(x => {
        val newLast = span(x).render
        domElement.parentElement.replaceChild(newLast, domElement)
        domElement = newLast
      })

    domElement
  }

  /**
    * Converts Behaviour that is used in the
    * context of a attribute value in scalatags to
```

```
    * an actual attribute value
    *
    * @return AttrValue
    */
  implicit def BehaviourToAttrValue[T: AttrValue] =
    new AttrValue[Behaviour[T]] {
      def apply(t: Element, a: Attr, r: Behaviour[T]): Unit = {
        r.changes()
          .listen((newVal) => {
            implicitly[AttrValue[T]].apply(t, a, newVal)
          })
      }
    }

  /**
    * Converts Behaviour that is used in the
    * context of a style value in scalatags to
    * an actual style value
    *
    * @return StyleValue
    */
  implicit def BehaviourToStyleValue[T: StyleValue] =
    new StyleValue[Behaviour[T]] {
      def apply(t: Element, s: Style, r: Behaviour[T]): Unit = {
        r.changes()
          .listen((newVal) => {
            implicitly[StyleValue[T]].apply(t, s, r.sampleNoTrans())
          })
      }
    }
}
```

A.3 Examples

A.3.1 Main

```
import rhein._
import rhein.ui._
import scala.scalajs.js.annotation.JSExportTopLevel
import scala.scalajs.js
import org.scalajs.dom
import org.scalajs.dom.html
import org.scalajs.dom.{Element}
import dom.document
import scalatags.JsDom.all._
```

```scala
import org.scalajs.dom.{Event => DomEvent}
import java.{util => ju}

import examples.todoApp.TodoApp
import examples.gameOfLife.GameOfLife

import scala.collection.mutable

// Exports and runs this main method in the javascript
// code that is generted by "fastOptJS"
@JSExportTopLevel("Main")
object Main {
  import Bindings._

  def main(args: Array[String]) {

    //Rhein
    document.body.appendChild(
      div(cls := "my-5")(
        img(src:= "https://i.imgur.com/2hb1EXu.png", style:= "width: 150px;
        ↪  margin-left: -5px"),
        h1( style :="font-size: 60pt")("Rhein"),
        p("Rhein is a data-propagation library based on Functional Reactive
        ↪  Programming abstractions such as Events and Behaviours that helps
        ↪  you to develop interactive applications using a
        ↪  conceptual-declarative approach that brings numerous benefits to
        ↪  the quality of the appli- cations and also solves several problems
        ↪  the mainstream methods of development of this type of software
        ↪  produce."),
      ).render
    )

    // Examples (applications + Example of Primitives)

    // 1. Applications
    TodoApp.run()
    var game = new GameOfLife()
    game.run(new mutable.ListBuffer(), true)

    // 2. Primitives in action

    // Label that always shows the current text
    val textField2: TextField = new TextField("Hello!")
    val label: Label = new Label(textField2.text)

    dom.document.body.appendChild(
```

```
    div(cls := "my-5")(
      h1("Label and Textfield"),

        div(cls := "mb-1")(
          span("Using: "),
          span(cls :="badge badge-info mr-1")("hold"),
          span(cls :="badge badge-secondary mr-1")("Label"),
          span(cls :="badge badge-secondary mr-1")("TextField")
        ),

        p(cls:= "w-75", "The hold primitive converts a event stream into a behaviour in such way that the behaviour's value is that of the most recent event received. The Label component corresponds to a simple p element that wraps the value of a behaviour and re-renders whenever the value of behaviour changes. The TextField component corresponds to an input(type = \"text\") element."),

        div(cls := "d-flex align-items-center")(
        textField2.domElement,
        span((cls := "ml-2"))(label.domElement),
      )
    ).render
  )

  // Using map to reverse string
  val textField3: TextField = new TextField("Hello!")
  val label3: Label = new Label(
    textField3.text.map((text => text.toString.reverse))
  )

  dom.document.body.appendChild(
    div(cls := "my-5")(
      h1("Using map to Reverse"),

      div(cls := "mb-1")(
          span("Using: "),
          span(cls :="badge badge-info mr-1")("hold"),
          span(cls :="badge badge-info mr-1")("map"),
          span(cls :="badge badge-secondary mr-1")("Label"),
          span(cls :="badge badge-secondary mr-1")("TextField")
        ),

        p(cls:= "w-75", "The map primitive is used to convert a stream of events of type A into a stream of events of type B by passing a function as argument that does the transformation"),
```

```

    div(cls := "d-flex align-items-center")(
      textField3.domElement,
      span((cls := "ml-2"))(label3.domElement)
    )
  ).render
)

// Merge example
val buttonA: Button = new Button("A", "btnA")
val buttonB: Button = new Button("B", "btnB")
val merged: Event[String] = Event.merge(
  buttonA.eventClicked.map(u => "A"),
  buttonB.eventClicked.map(u => "B")
)
val textField4: TextField = new TextField(merged, "")

dom.document.body.appendChild(
  div(cls := "my-5")(
    h1("Merge example"),

    div(cls := "mb-1")(
        span("Using: "),
        span(cls :="badge badge-info mr-1")("merge"),
        span(cls :="badge badge-info mr-1")("map"),
        span(cls :="badge badge-secondary mr-1")("TextField"),
        span(cls :="badge badge-secondary mr-1")("Button")
      ),

      p(cls:= "w-75", "The merge primitive puts the event firings from two
      ↪  event streams together into a single stream. This function is
      ↪  semantically equivalent to the one we apply to collections (i.e.
      ↪  lists). The Button component corresponds to a div element that
      ↪  is attached with an onClick DOM listener."),

    div(cls := "d-flex align-items-center")(
      buttonA.domElement,
      buttonB.domElement,
      span((cls := "ml-2"))(textField4.domElement)
    )
  ).render
)

// Merge and Hold examples
val buttonRed: Button = new Button("Red", "btn-danger")
val buttonGreen: Button = new Button("Green", "btn-success")
```

```scala
val buttonsMerged: Event[String] = Event.merge(
  buttonRed.eventClicked.map(_ => "Red"),
  buttonGreen.eventClicked.map(_ => "Green")
)
val labelRedOrGreen: Label = new Label(buttonsMerged.hold(""))

dom.document.body.appendChild(
  div(cls := "my-5")(
    h1("Merge and hold example"),

    div(cls := "mb-1")(
        span("Using: "),
        span(cls :="badge badge-info mr-1")("merge"),
        span(cls :="badge badge-info mr-1")("map"),
        span(cls :="badge badge-info mr-1")("hold"),
        span(cls :="badge badge-secondary mr-1")("Label"),
        span(cls :="badge badge-secondary mr-1")("Button")
      ),

      br,

    div(cls := "d-flex align-items-center")(
      buttonRed.domElement,
      buttonGreen.domElement,
      span((cls := "ml-2"))(labelRedOrGreen.domElement)
    )
  ).render
)

// Snapshot
val translateButton: Button = new Button("Translate", "btnTranslate")
val english: TextField = new TextField("Translate")
val snapshotVal: Event[String] = translateButton.eventClicked.snapshot(
  english.text,
  (u, txt: String) => txt.trim.replaceAll(" |$", "us ").trim
)
val latin: Label = new Label(snapshotVal.hold(""))

dom.document.body.appendChild(
  div(cls := "my-5")(
    h1("Shanpshot"),

    div(cls := "mb-1")(
        span("Using: "),
        span(cls :="badge badge-info mr-1")("snapshot"),
        span(cls :="badge badge-info mr-1")("hold"),
```

```
        span(cls :="badge badge-secondary mr-1")("TextField"),
        span(cls :="badge badge-secondary mr-1")("Button")
      ),

      p(cls:= "w-75", "The snapshot primitive captures the value of a behaviour at the time when an event stream fires, and then it can combine the payload from the event stream and the one from the behaviour together with a supplied function."),


    div(cls := "d-flex align-items-center")(
      translateButton.domElement,
      english.domElement,
      span((cls := "ml-2"))(latin.domElement)
    )
  ).render
)

// Accuulator - Spinner example
val valueLoop: BehaviourLoop[Int] = new BehaviourLoop()

val buttonPlus: Button = new Button("+", "btnPlus")
val buttonMinus: Button = new Button("-", "btnMinus")
val sDelta: Event[Int] = Event.merge(
  buttonPlus.eventClicked.map(_ => 1),
  buttonMinus.eventClicked.map(_ => -1)
)
val sUpdate: Event[Int] =
  sDelta
    .snapshot(valueLoop, (delta, value_ : Int) => {
      val res = value_ + delta
      res
    })
    .filter(n => n >= 0)
valueLoop.loop(sUpdate.hold(0))
val resultLabel: Label = new Label(valueLoop.map(x => x.toString()))
dom.document.body.appendChild(
  div(cls := "my-5")(
    h1("Accumulator"),

    div(cls := "mb-1")(
        span("Using: "),
        span(cls :="badge badge-info mr-1")("loop"),
        span(cls :="badge badge-info mr-1")("hold"),
        span(cls :="badge badge-info mr-1")("merge"),
        span(cls :="badge badge-info mr-1")("map"),
```

```scala
        span(cls :="badge badge-info mr-1")("filter"),
        span(cls :="badge badge-secondary mr-1")("Label"),
        span(cls :="badge badge-secondary mr-1")("Button")
      ),

      p(cls:= "w-75", "An Accumulator represents a state that is updated by combining new information with the existing state."),

    div(cls := "d-flex align-items-center")(
      buttonPlus.domElement,
      buttonMinus.domElement,
      span((cls := "ml-2"))(resultLabel.domElement)
    )
  ).render
)

// Lifting example
val textFieldA: TextField = new TextField("0")
val textFieldB: TextField = new TextField("0")
val a: Behaviour[Int] = textFieldA.text.map(t => t.toInt)
val b: Behaviour[Int] = textFieldB.text.map(t => t.toInt)
// def add(a: Int, b: Int): Int = a + b
val lifted = a.lift(b, (p, q: Int) => p + q)
val res: Label = new Label(lifted.map(x => x.toString))

dom.document.body.appendChild(
  div(cls := "my-5")(
    h1("Lift"),

    div(cls := "mb-1")(
        span("Using: "),
        span(cls :="badge badge-info mr-1")("lift"),
        span(cls :="badge badge-info mr-1")("map"),
        span(cls :="badge badge-secondary mr-1")("Label"),
        span(cls :="badge badge-secondary mr-1")("TextField")
      ),

      p(cls:= "w-75", "The lift primitive allows you to combine two or more behaviours into one using a specified combining function. The filter primitive is used to let event stream values through the pipe only sometimes. This is a general functional programming concept, and this name is used universally in FRP systems."),

    div(cls := "d-flex align-items-center")(
      textFieldA.domElement,
```

```
          textFieldB.domElement,
          span((cls := "ml-2"))(res.domElement),

        ),
        div(cls := "alert alert-light my-5", role := "alert", "All examples are implemented in Rhein")
      ).render
    )

  }
}
```

A.3.2 TodoApp

```
package examples.todoApp
import rhein._
import rhein.ui._
import scalatags.JsDom.all._
import scala.scalajs.js._
import scala.scalajs.js.annotation._
import org.scalajs.dom
import org.scalajs.dom._

/**
  * Value that will be emitted
  * when emitting events in the todo application
  * @param action can either be add or remove
  * @param value will specify which value in the list will the action will be taken on
  */
class TodoMessage[T](var action: String, var value: T) extends Message

/**
  * Simmple Task class to model a task
  * @param description
  * @param isDone
  */
class Task(var description: String, var isDone: Boolean) {
  val id: String = java.util.UUID.randomUUID.toString
}

/**
  * Todo application
  */
object TodoApp {
  import Bindings._
```

```scala

/**
  * UI component for a Task
  * CSS classes are part of Bootstrap v4.0
  * @param task
  */
def todoItem(task: Task, buttonDelete: Button) = {
  div(cls := "d-flex align-items-center mb-1")(
    div(cls := "d-flex align-items-center border p-2")(
      span(cls := "mr-2", task.description),
      if (!task.isDone)
        span(cls := "badge badge-pill badge-warning", "Ongoing")
      else span(cls := "badge badge-pill badge-success", "Done")
    ),
    buttonDelete.domElement(cls := "btn btn-light ml-2")
  )
}

/**
  * Method to initialise and run
  * the todo application.
  */
def run() {
  // UI components
  val buttonAdd = new Button("+")
  val taskTextField = new TextField("")

  /**
    * Stream of events
    * When the add button is clicked, a new event is
    */
  val todoEvents = buttonAdd.eventClicked
    .snapshot(taskTextField.text, (click, text: String) => {
      new TodoMessage[Task]("add", new Task(text, false))
    })

  // Loop State
  // Creating an accumulator
  val cState = new BehaviourLoop[List[Task]]
  val updates = todoEvents.snapshot(cState, (event, _state: List[Task]) => {
    event.action match {
      case "add" => {
        event.value :: _state
      }
      case "remove" => {
        _state.filter(p => p.id != event.value.id)
```

```
        }
      }
    })
    cState.loop(updates.hold(List()))

    // Displaying the Todos
    val list = new Listing[Task](cState, (task: Task, index: Int) => {
      val buttonDelete = new Button("Mark as done!")
      buttonDelete
        .attachEvent(todoEvents, new TodoMessage[Task]("remove", task))
      todoItem(task, buttonDelete)
    })

    //Rendering
    dom.document.body.appendChild(
      div(cls := "mt-2")(
        h1("Todo Application"),
        p("Please add a new task"),
        div(cls := "d-flex align-items-center")(
          buttonAdd.domElement,
          taskTextField.domElement
        ),
        br,
        br,
        list.domElement
      ).render
    )

  }
}
```

A.3.3 GameOfLife

```
package examples.gameOfLife

import rhein.Behaviour
import rhein._
import rhein.ui.{Listing, Button}
import scalatags.JsDom.all._
import scala.scalajs.js._
import org.scalajs.dom.{Event => DomEvent}
import org.scalajs.dom
import scala.collection.mutable.ListBuffer
import scalatags.JsDom
import scala.io.Source
import scala.util.Random
```

```
import java.{util => ju}

/**
  * Class representing Game Of Life
  * To run the game, please create a GameOfLife Object and
  * call the run() method
  *
  */
class GameOfLife {
  type World = ListBuffer[Boolean]

  var pattern1 = ListBuffer(
    ListBuffer(0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0),
    ListBuffer(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),
    ListBuffer(1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1),
    ListBuffer(1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1),
    ListBuffer(1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1),
    ListBuffer(0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0),
    ListBuffer(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),
    ListBuffer(0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0),
    ListBuffer(1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1),
    ListBuffer(1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1),
    ListBuffer(1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1),
    ListBuffer(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),
    ListBuffer(0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0)
  )

  var pattern2 = ListBuffer(
    ListBuffer(1, 1, 0, 0),
    ListBuffer(1, 1, 0, 0),
    ListBuffer(0, 0, 1, 1),
    ListBuffer(0, 0, 1, 1)
  )

  // The speed of transition between states
  val INTERVAL = 100
  var WIDTH: Int = 30
  var HEIGHT: Int = 30

  // choose a pattern between the two provided
  // if the second parameter == true => random
  //run(pattern1, true)

  // Uncomment this line to run the simulation
  // witouth FRP. Don't forget to comment the method above in order to
  // prevent running two simulations at the same time.
```

```scala

//runWithouthFRP()

// Run method that starts the game of life
def run(initialPattern: ListBuffer[ListBuffer[Int]], random: Boolean) {
  // Configuration variables
  var world: World = new World()

  if (random) {
    world = generateWorld(WIDTH, HEIGHT)
  } else {
    WIDTH = initialPattern.length
    HEIGHT = initialPattern(0).length

    world = createWorld(initialPattern)
  }

  // Event stream that emits a one Unit event at interval specified
  var tickStream: Event[Unit] = Event.interval(INTERVAL, INTERVAL)

  // UI pause button
  var buttonPause: Button = new Button("Pause")
  // Holds the state paused or not
  var pauseLoop: EventLoop[Boolean] = new EventLoop()
  var pauseState = pauseLoop.hold(true)
  pauseLoop.loop(
    buttonPause.eventClicked
      .map(click => true)
      .snapshot(pauseState, (event, _pauseState: Boolean) => {
        !_pauseState
      })
  )

  // Filterin out tick events when we are in pause state
  tickStream = tickStream.filter(x => pauseState.sampleNoTrans())

  /**
    * Holds the state of the world
    * The datatype of EventLoop is List[World] but it could just be World.
    * The decision of wrapping the World in a List is to facilitate the use
    * of Listing UI Component
    */
  var eventLoop: EventLoop[List[World]] = new EventLoop()
  var cState = eventLoop.hold(List(world))

```

```
    eventLoop.loop(tickStream.snapshot(cState, (event, _state: List[World]) => {
      List(updateWorld(_state.head, WIDTH, HEIGHT))
    }))

    // UI component for the grid - it holds the current state of the world
    // and maps the world to a DOM table/grid with the provided function
    var grid: Listing[World] =
      new Listing(cState, (world: World, index: Int) => {
        makeGrid(world)
      })

    // Appending all UI components to the DOM
    dom.document.body.appendChild(
      div(cls := "mt-3")(
        h1("Game of Life"),
        grid.domElement,
        br,
        buttonPause.domElement
      ).render
    )
  }

  /**
    * UI Component representing a single cell
    * If cell is alive, returns a black square, otherwise white square
    * @param alive
    * @return td HTML fragment
    */
  def cellTd(alive: Boolean) = {
    var color = if (alive) "#000" else "#fff"
    td(
      style := s"width: 15px; height: 15px; border: 0.5px solid #ccc; background-color: ${color};"
    )
  }

  /**
    * UI Component representing the whole World / grid
    * It maps through all cells in the World and
    * creates a square according to the cell state
    * @param world
    * @return table HTML fragment
    */
  def makeGrid(world: World) = {
    table(style := "border-collapse: collapse")(
```

```scala
      tbody(
        world
          .sliding(HEIGHT, HEIGHT)
          .toList
          .map((row: ListBuffer[Boolean]) => {
            tr(row.map((cell: Boolean) => {
              cellTd(cell)
            }))
          })
      )
    )
  }

  /**
    * Generate a random world given the size
    * @param width
    * @param height
    * @return
    */
  def generateWorld(width: Int, height: Int) = {
    val r = new ju.Random()
    val newWorld: World = ListBuffer.fill(WIDTH * HEIGHT)(false)
    for (y <- 0 until width) {
      for (x <- 0 until height) {
        val rand = r.nextFloat()
        newWorld(y * WIDTH + x) = if (rand < 0.5) false else true

      }
    }
    newWorld
  }

  /**
    * Create World state given an initial pattern / configuration for
    * the map / world
    * @param initial  Initial pattern
    * @return World state with initial pattern
    */
  def createWorld(initial: ListBuffer[ListBuffer[Int]]) = {
    val newWorld: World = ListBuffer.fill(WIDTH * HEIGHT)(false)

    for (y <- 0 until initial.length) {
      for (x <- 0 until initial(0).length) {
        if (y < HEIGHT && x < WIDTH) {
          newWorld(y * WIDTH + x) = (initial(y)(x) == 1);
        }
```

```
      }
    }
    newWorld
  }

  /**
    * Run the simulation without FRP
    *
    * @param initialPattern
    */
  def runWithouthFRP(initialPattern: ListBuffer[ListBuffer[Int]]) {

    var worldState = createWorld(initialPattern)
    var activeState = true;

    val root = dom.document.createElement("div")
    root.id = "root"

    var gridNode = makeGrid(worldState).render

    val pauseButton = dom.document.createElement("button")
    pauseButton.addEventListener("click", (ev: DomEvent) => {
      activeState = !activeState
    })

    root.appendChild(gridNode)

    dom.document.body.appendChild(root)
    dom.document.body.appendChild(pauseButton)

    scala.scalajs.js.timers.setInterval(INTERVAL) {
      val t0 = System.nanoTime()

      if (activeState) {
        worldState = updateWorld(worldState, WIDTH, HEIGHT);
        var grid = makeGrid(worldState).render
        dom.document.getElementById("root").replaceChild(grid, gridNode)
        gridNode = grid
      }
      val t1 = System.nanoTime()
      println(t1 - t0)
    }

  }

  // FRP Logic
```

```scala
// REF! - Functions based on
// https://github.com/gabriellesc/FRP-intro/blob/master/FRP-GOL-init/app.js

/**
  * Returns true if cell at position x, y is alive, false otherwise
  *
  * @param x
  * @param y
  * @param world
  * @param width
  * @param height
  * @return Cell's' current state
  */
def isAlive(x: Int, y: Int, world: World, width: Int, height: Int) = {
  x >= 0 && y >= 0 && x < width && y < height && world(y * width + x)
}

/**
  * Returns number of neighbours around a cell at position x y
  *
  * @param x
  * @param y
  * @param world
  * @param width
  * @param height
  * @return Nb of neighbours
  */
def getNumberOfNeighbours(
    x: Int,
    y: Int,
    world: World,
    width: Int,
    height: Int
) = {
  var total = 0
  if (isAlive(x, y + 1, world, width, height)) total = total + 1
  if (isAlive(x - 1, y + 1, world, width, height)) total = total + 1
  if (isAlive(x + 1, y + 1, world, width, height)) total = total + 1
  if (isAlive(x, y - 1, world, width, height)) total = total + 1
  if (isAlive(x - 1, y - 1, world, width, height)) total = total + 1
  if (isAlive(x + 1, y - 1, world, width, height)) total = total + 1
  if (isAlive(x - 1, y, world, width, height)) total = total + 1
  if (isAlive(x + 1, y, world, width, height)) total = total + 1
  total
}
```

```
/**
  * Update cell state at position x y
  * given the standard game of life rules
  *
  * @param x
  * @param y
  * @param world
  * @param width
  * @param height
  * @return Updated world with cell at x, y changed
  */
def updateCellState(
    x: Int,
    y: Int,
    world: World,
    width: Int,
    height: Int
): Boolean = {
  val numberOfNeighbours = getNumberOfNeighbours(x, y, world, width, height)
  if (numberOfNeighbours < 2 || numberOfNeighbours > 3)
    return false
  else if (numberOfNeighbours == 3) {
    return true;
  }
  return world(y * width + x)
}

/**
  * Updates all cells in the world
  *
  * @param world
  * @param width
  * @param height
  * @return New state for the world
  */
def updateWorld(world: World, width: Int, height: Int): World = {
  /* make a copy of world to modify */
  var newWorld = world.clone()

  for (y <- 0 until height) {
    for (x <- 0 until width) {
      newWorld(y * width + x) = updateCellState(x, y, world, height, width);
    }
  }

  newWorld
```

```
  }

}
```

A.4 Unit Tests

A.4.1 RheinTester

```
package rhein
import org.junit.After
import org.junit.Assert.assertEquals
import org.junit.Test
import scala.collection.mutable.ListBuffer

class RheinTester {

  @Test
  def testSendEvent(): Unit = {
    val event = new EventSink[Int]()
    val out = new ListBuffer[Int]()
    val listener = event.listen(x => out += x)
    event.send(1)
    listener.unlisten()
    assertEquals(List(1), out)
    event.send(2)
    assertEquals(List(1), out)
  }

  @Test
  def testMapPrimitive(): Unit = {
    val event = new EventSink[Int]()
    val map = event.map(x => x.toString)
    val out = new ListBuffer[String]()
    val listener = map.listen(x => out += x)
    event.send(5)
    listener.unlisten()
    assertEquals(List("5"), out)
  }

  @Test
  def testMergePrimitive(): Unit = {
    val e1 = new EventSink[Int]()
    val e2 = new EventSink[Int]()
    val out = new ListBuffer[Int]()
    val listener = Event.merge(e2, e1).listen(x => out += x)
```

```scala
    e1.send(1)
    e2.send(2)
    e1.send(3)
    listener.unlisten()
    assertEquals(List(1, 2, 3), out)
  }

  @Test
  def testFilterPrimitive(): Unit = {
    val event = new EventSink[Char]()
    val out = new ListBuffer[Char]()
    val listener =
      event.filter(c => Character.isUpperCase(c)).listen(x => out += x)
    List('H', 'o', 'I').foreach(event.send(_))
    listener.unlisten()
    assertEquals(List('H', 'I'), out)
  }

  @Test
  def testLiftPrimitive(): Unit = {
    val out = new ListBuffer[Int]()
    val behaviourSink = new BehaviourSink(Some(1))
    val cell = behaviourSink.map(v => 2 * v)
    val listener = behaviourSink
      .lift(cell, (x: Int, y: Int) => x + y)
      .changes()
      .listen(x => out += x)
    behaviourSink.send(2)
    behaviourSink.send(7)
    listener.unlisten()
    assertEquals(List(6, 21), out)
  }

  @Test
  def testHoldPrimitive(): Unit = {
    val event = new EventSink[Int]()
    val behaviour = event.hold(0)
    val out = new ListBuffer[Int]()
    val listener = event.listen(x => out += x)
    List(2, 9).foreach(event.send)
    listener.unlisten()
    assertEquals(List(2, 9), out)
  }

  @Test
  def testSnapshotPrimitive(): Unit = {
```

```
    val behaviourSink: BehaviourSink[Int] = new BehaviourSink(Some(0))
    val event = new EventSink[Long]()
    val out = new ListBuffer[String]()
    val listener = event
      .snapshot[Int, String](behaviourSink, (x: Long, y: Int) => x + " " + y)
      .listen(x => out += x)
    event.send(100L)
    behaviourSink.send(2)
    event.send(200L)
    behaviourSink.send(9)
    behaviourSink.send(1)
    event.send(300L)
    listener.unlisten()
    assertEquals(List("100 0", "200 2", "300 1"), out)
  }

}
```

A.5 Project Config

A.5.1 build.sbt

```
enablePlugins(ScalaJSPlugin)
enablePlugins(ScalaJSJUnitPlugin)

scalaVersion := "2.12.8"

libraryDependencies += "com.lihaoyi" %%% "scalatags" % "0.7.0"
libraryDependencies += "org.scala-js" %%% "scalajs-dom" % "0.9.7"
libraryDependencies += "com.novocode" % "junit-interface" % "0.11" % "test->default"
libraryDependencies += "junit" % "junit" % "4.12" % "test"

// This is an application with a main method
scalaJSUseMainModuleInitializer := true
```

A.5.2 plugins.sbt

```
addSbtPlugin("org.scala-js" % "sbt-scalajs" % "0.6.31")
```

A.5.3 build.properties

```
sbt.version=1.3.2
```

Appendix B

User Guide

B.1 Instructions

Note! The package is not deployed to any dependency management server. You will have to add the `rhein` folder inside your project to use Rhein.

1. After you've successfully installed sbt on your machine, go to the root directory of the project and run `sbt`. This process takes a bit longer for the first run, because it downloads all dependencies.

2. After you have sucessfully launeched the build tool, you now have to choose what you wnat to do. You can either compile the current examples using `fastOptJS` which generates a javascript file in `target/scala-2.12/rhein-fastopt.js`. You can add this file in a HTML file, but we already provided a file that has this file linked in `src/main/resources/index-opt.html` To run, open this file in a browser.

3. If you decide to create new code, you must import the package using `import rhein._`. This imports all FRP abstractions available in Rhein. To import the UI Binding library use `import rhein.ui._`. PS! If you want to use the Bindings from the UI Biding Library, you neeed to import this file in your class using `import Bindings._` (make sure you have the ui package imported first).

4. If you want to render elements in the browser, you must import scalatags and scalajs. The main entry of the program is the `Main.scala` class. You can change this to any class you want, but make sure you specify which is the main entry in the compiled javascript using `@JSExportTopLevel("Main")`.

5. To run tests, type `testOnly` in your sbt.

B.2 Requirements

The following are required to run Rhein

- Scala 2.12.8
- sbt 1.3.2

Rhein uses these dependencies which are already specified in the `build.sbt` file:

- scala-js 0.6.31
- scalajs-dom 0.9.7
- scalatags 0.7.0

B.3 Links

Github Repository

Deployment of the examples