Experience Report: Functional Reactive Programming and the DOM

Bob Reynders  Dominique Devriese  Frank Piessens
imec - DistriNet - KU Leuven
{firstname.lastname}@cs.kuleuven.be

Abstract

Web applications are inherently event-driven and traditionally implemented using imperative callbacks in Javascript. An alternative approach for such programs is functional reactive programming (FRP). FRP offers abstractions to make event-driven programming convenient, safe and composable, but like pure functions it is isolated from the 'outside' world. In this paper we describe our experience in developing a library that binds FRP to the document object model (DOM). We describe that in its current state there are fundamental issues that do not yet have a perfect solution. We expand upon the functionality of existing FRP DOM libraries with an FRP model for DOM properties. We show that despite of some design problems a pragmatic library can be created that can be used to create web applications.

1. Introduction

Web applications are inherently event-driven and traditionally implemented using imperative callbacks and mutable state. An alternative approach to writing such programs is functional reactive programming (FRP). It offers abstractions to make event-driven programming convenient, safe and composable and it has been successfully applied to the web before [1, 7].

FRP has two main primitives: events (a stream of values at discrete times) and behaviors (time-varying values). An FRP application is constructed by composing behaviors and events with a set of FRP operations. However, a program that is only made out of FRP components is like a pure function. For it to be useful it has to be able to interact with the 'outside' world. One of those interactions is with the document object model (DOM). In this paper we discuss non-trivial design problems when defining an FRP library for the DOM that do not yet have ideal solutions such as:

**Type of Main?** Entry points of programs are traditionally imperative main functions. In this paper we focus on a declarative alternative, a natural approach for FRP programs. We see two design choices. In the first approach, main has type $\text{Html}$ where $\text{Html}$ represents a static HTML tree that may contain dynamic parts, i.e., there are APIs with types like $\text{p: Behavior[Html]} \Rightarrow \text{Element}$ to build elements with varying context. In the second approach, main has type $\text{Behavior[Html]}$ where the type $\text{Html}$ now represents an entirely static DOM tree. For example, an application that displays a text behavior:

$$\begin{align*}
\text{val text : Behavior[String]} &= \ldots \\
\text{val main : Html} &= \text{p(text)} \\
&// \text{ or} \\
\text{val main : Behavior[Html]} &= \text{text.map(s \Rightarrow p(s))}
\end{align*}$$

In the first case the application is a static p-element with dynamic content of type $\text{Behavior[String]}$. In the second case the application is defined as a behavior of a completely static p-element with content of type $\text{String}$. In section 3, we discuss both choices and their consequences in depth.

**Recursion between FRP and the DOM.** Irrespective of the above choice, the DOM tree in an FRP application can change in response to changes in FRP behaviors and events. However, such new elements in the DOM tree may also produce new primitive event streams that represent, for example, button clicks on the event, which the FRP program needs to be able to react to. In other words, there is a cycle of dependencies between the FRP program’s behaviors and events, the DOM tree that they define ($\text{main}$) and the primitive behaviors for elements in this DOM tree (e.g., click events on buttons).

For example (using the $\text{Behavior[Html]}$ approach of $\text{main}$), we create a div that contains two text inputs. The first text input always mirrors the contents of the second:

$$\begin{align*}
\text{val content : Behavior[String]} &= \ldots \\
\text{val main} &= \text{content.map (str \Rightarrow} \\
&\text{div(\text{id := "one", tpe := text}),} \\
&\text{\text{input(\text{id := "two", tpe := text, str.reverse}))}
\end{align*}$$

How do we define $\text{content}$? This is a recursive problem and the solution to it affects both the purity and the usability of the library.

**Pull-based Interfaces to the DOM.** When linking the API to the DOM it is important to think about the underlying FRP library’s evaluation strategy.

The DOM produces events and invokes handlers to represent actions such as user interactions. But, other than an event producer, the DOM is also a queryable source of data with for example elements that contain multiple mutable fields such as css classnames or a user’s input.

Current FRP DOM libraries focus entirely on events and push-driven evaluation. Reading values from input elements in existing libraries is mimicked by listening to the appropriate events and rebuilding an in-FRP representation of the actual value. Not only does this create overhead, it makes the correctness of a behavior rely on the ability to detect all changes. These libraries for example do not propagate changes that are made through Javascript property assignments such as:

```javascript
document.getElementById("field").value = "123"
```
This hinders common web-development practices such as enhancing existing form elements with datepickers or autocompletion engines by adding small Javascript libraries. The developer is forced to forward the added script’s events to the FRP system to maintain correctness. In this case, there is actually a known solution that does not appear to have many downsides, namely push-pull FRP [3]. However, we know of no system that applies this solution in an FRP web framework, so it seems worth discussing here.

1.1 Outline

In this paper we report on our experiences gained while defining an FRP library for the DOM. The library is being developed as part of a project in which FRP is used as multi-tier web-application paradigm, a successor to the ideas that were described in [8]. We identify non-trivial design problems that often do not have ideal solutions. We start with a quick introduction to FRP in section 2. Next, we go over the main design problems we encountered while developing our library. The type of main in section 3, recursion between FRP and the DOM in section 4 and interfacing with the DOM and non-FRP code in section 5.

While explaining the design problems and their possible solutions we use an FRP DOM implementation of a to-do list manager as an example. Its functionality is simple and a screenshot is shown in fig. 1, a user can add entries and an entry consists of some content and a deadline. All API definitions and code examples are written in Scala but do not use exotic features that would make it hard to port to a different language.

In section 6, we finish the to-do list manager example and demonstrate our final API. We conclude in section 7 and discuss related work in section 8.

2. Functional Reactive Programming

We start off with a small introduction to our underlying FRP library. Let us start by going over the FRP primitives, event and behavior:

**Events** can be seen as a sequence of discrete values. Common examples of events are mouse clicks or button presses since they are occurrences that can be timestamped. There are three core operations on events: map, filter and merge as shown in fig. 2. We do not discuss map or filter since they behave just like their well-known collection counterparts. merge takes two events (of the same type) and returns an event that fires whenever one of the original events fire. If both events fire at the same time, the given function combines both values into a single new one.

**Behaviors** are values that vary continuously over time. An example of a behavior is the position of the cursor. A mouse is always somewhere but its position may change continuously as you move your hand. The two core operations on behaviors are: map2 and constant as shown in fig. 2. Constant creates a behavior that never changes its value. map2 has the ability to combine two behaviors with a function. Other convenience functions such as map can be defined in terms of constant and map2.

**Discrete Behaviors.** While behaviors can theoretically change continuously, they often change at specific times. In these cases we know more about them, we know when they change. This extra information can be exposed to the programmer through a specific interface of discrete behaviors. Other than exposing the time at which discrete behaviors change (def changes: Event[A]), their API is identical to that of continuous behaviors.

Throughout this paper we use an FRP library that has both the discrete (DBehavior) and continuous behaviors (CBehavior) primitives available separately.

**Events ⇔ Behaviors** Converting from events to behaviors and vice-versa is done through two other operations: Event.fold and Behavior.snapshot, also shown in fig. 2. Folding an event is similar to folding a list, a starting value and an accumulation function is given to compute a new value whenever a new element arises. Its result is a behavior representing the ongoing accumulation. This behavior changes discretely whenever the event fires. Snapshotting a behavior with an event allows you to inspect the value of a behavior at the rate of that event. The behavior is sampled for every change in the event by applying a combination function to the event value and the behavior’s value at the time. A simpler version of snapshot called sampledBy ignores the event value and simply samples the behavior at the appropriate times.

**Design Decisions & Properties.** The FRP library that we use throughout the paper and its semantics make a couple of design decisions that differ from other FRP libraries. Most importantly, both discrete and continuous behaviors co-exist as different primitives
exposing the push-pull evaluation explicitly as in [5] unlike the push-pull library from [3]. The library is also entirely first-order, it offers no methods to flatten nested FRP primitives, and this paper and its proposed DOM library do not require them. This makes the FRP library less expressive, but prevents problems with time leaks and makes other primitives such as replication easier to implement.¹

3. Type of Main?

The traditional entry point of an application is an imperative main function or some given code block. In this section we only focus on declarative alternatives as an entry point.

We describe two designs choices that can be made for a declarative main function and discuss their differences. The examples in this paper use an element constructor library called ‘scalatags’². Instead of writing XML tags to represent elements we use Scala functions with the appropriate name. The DSL supports attributes and children values, for example:

```
<div id="content">Hello World</div>
```

We become:

```
div(id := "content", p("Hello World"))
```

3.1 Html.

The first design represents main as a static HTML document. This document defines the entire application and dynamic behavior is embedded within. We give it the type Html. We take the todo-list application’s interface as an example:

```
case class Entry(content: String, date: String)
def template(entry: Entry): Html = ...  
val state: DBehavior[List[Entry]] = ...
val inputForm: Html = ...  
val main: Html = div(h1("Todo!"), inputForm, h2("Entries"), div(state.map(_.map(template))))
```

We model the to-do entry as a case class with two fields and assume a function template that takes an entry and returns an element that properly lays out its contents. state is a discrete behavior of entries, that is, a discretely changing to-do list. inputForm is the form from fig. 1 that contains the text inputs and buttons required to enter entries for the application. Neither implementations matter here, we leave them undefined. With these components we define the main value as a div with a top title, an input form, a subtitle and a list of all the current entries. The element constructor functions have the following type³:

```
def 'tag'(attributes: AttrPair*, children: (Html || DBehavior[List[Html]])*): Html
```

An element is created using its corresponding tag function, a variable amount of attributes can be set and children elements can be added. Children can be of two types, the first is a simple Html value. It indicates a static amount of children, for example, a tag that is created with three Html children always has three children during the runtime of the program. The second type of children, DBehavior[List[Html]], represents the dynamic pieces in the static Html document. It allows for children to be defined as a discrete behavior of elements, a collection that can change its size throughout the runtime of the program. In our example this is shown through the use of div(entries.map(_.map(template))) which creates a discrete behavior with a list of layouted entries.

Required: Higher-order FRP. We rely on modeling element children with behaviors to support a dynamic amount of elements in this type of DSL. However, this implies a higher-orderness of the solution, for example, we can define an element that contains a discrete behavior of elements which in turn contain more discrete behaviors of elements. Higher-order FRP is a tricky subject that cannot be implemented without either: (1) recomputing all previous values of any generated behavior or event, (2) making the first-order API more inconvenient (complicating fold etc.) [9] or (3) making the higher-order API more inconvenient by restricting which behaviors may be nested into others [6]. Each proposed solution has its own drawbacks in either usability or API flexibility.

Implementation. With a static document representation all dynamic elements are explicitly added. Dynamic behavior is only possible with embedded DBehavior elements. An implementation makes use of this information by attaching and detaching hooks to the DBehavior elements when needed. These hooks call into DOM APIs to create, delete and modify elements as needed.

3.2 Behavior of Html.

The alternative design has a discrete behavior as its main value. Instead of defining an application as a user interface element, we define the application as a discretely changing value of elements, i.e., DBehavior[Html]. In this case Html is completely static, all changes to the user interface are expressed through the use of DBehavior.

Let us look at the to-do list application again using the same definition for Entry, template, inputForm and state:

```
val main: DBehavior[Html] = state.map { entries ⇒  
val eList = div(entries.map(template))  
val main: DBehavior[Html] = state.map { entries ⇒
  val eList = div(entries.map(template))
  div(h1("Todo!"), inputForm, h2("Entries"), eList)
}
```

Instead of ‘embedding’ the state behavior in our user interface code, we now map over it. We take the state behavior and based on its entries we create the same interface as before. Showing the current list of entries in the program is now done by mapping template directly over entries. Note that the constructor functions in this design no longer take DBehavior[Html] elements, in a similar notation as before:

```
def 'tag'(atts: AttrPair*, children: Html*): Html
```

In this design, it is natural to express a variable amount of children since the main value itself declares that it is a discretely changing value.

First-order FRP. An advantage of structuring the type of main this way is that there is no need for a higher-order FRP primitive in the underlying FRP library. All programs are expressed through the regular behavior primitives such as map².

Implementation. Representing main as a behavior of Html requires a more advanced implementation. The only knowledge that main conveys is what the interface should look like at certain points in time. It does not define what changed between two interfaces. The implementation has to recover the changes and turn them into actual DOM operations. This phenomenon has gotten popular in practice under the term Virtual DOM, for example in Facebook’s React⁴.

¹ See, e.g., [9] for more details on time leaks.
² http://www.11haoyi.com/scalatags/
³ This is a simplified notation of this function, it is not valid Scala but it can be implemented in several ways, for example, a type-class based approach.
⁴ https://facebook.github.io/react/
4. Recursion between FRP and the DOM

There is no ‘best’ way to do foreign function interfaces (FFIs) in FRP, but most FRP libraries have an imperative creation of an event source, for example:

```scala
val source = Event.source[Int]
val acc = source.fold(0)(_ + _)
source.fire(2) // acc is now 2
source.fire(5) // acc is now 7
```

Regardless of the chosen FRP FFI input, there is an additional issue when describing GUIs. As we have seen, the entire application is defined in terms of the user interface with main for example being DBehavior[Html]. The problem is that the user interface also (albeit indirectly) defines the user’s input to the program. An interface contains forms that can be filled in, buttons that can be pressed, etc. This input is then used to define main itself which turns our GUI application into a recursive problem.

A clean solution to this problem could be to come up with a fixpoint primitive. We expect it to look similar to [2] but are not sure of its exact specifications. In this paper we focus on more pragmatic solutions that break the recursion by pre-declaring possible inputs. We expect it to look similar to [2] but are not sure of its exact specifications. In this paper we focus on more pragmatic solutions that break the recursion by pre-declaring possible inputs. We expect it to look similar to [2] but are not sure of its exact specifications. In this paper we focus on more pragmatic solutions that break the recursion by pre-declaring possible inputs.

**Deriving Events from Elements.** As usual we look to the to-do list application for an example, this time its input form and the corresponding submit event:

```scala
val messageInput: Html = ...
val dateInput: Html = ...
val inputForm = form(
  messageInput,
  dateInput,
  button(tpe := "submit", "Add")
)
val submit: Event[dom.Event] = inputForm.listen(_.onsubmit)
```

The input form is a simple HTML form with two inputs (messageInput and dateInput) and a submit button. Its creation is straightforward and uses the familiar element constructor DSL. Retrieving the submit event from the form is done by calling a listen method with a function that says which property you would like to add a callback too. In this case we attach to the onsubmit property and create an FRP event submit that propagates DOM events whenever the form is submitted.

This type of input retrieval has some implications on purity. Up to now the element constructor functions were **referentially transparent**:

```scala
val x = button()
val el1 = div(x, button())
val el2 = div(x, x)
```
el1 and el2 would be freely interchangeable, regardless of what code was added. If it becomes possible to derive events from elements, this can no longer be the case. Even though the definition of x is exactly the same as button(), it would still be observable as a different instance through events:

```scala
val x = button()
val el1 = div(x, button())
val el2 = div(x, x)
val clicks = x.listen(_.onclick)
```

When el1 is embedded in the user interface the event clicks would produce values when the first button was clicked. However, if el2 is embedded in the interface, clicks would produce values when either of the buttons were clicked.

While element constructors become impure, the listen method stays referentially transparent since all calls to the same instance return the same event. We trade one point of referential transparency for the other.

#### Creating Elements with Event Sources.

A second implementation of the to-do list’s input form and submit event makes use of event sources:

```scala
val messageInput: Html = ...
val dateInput: Html = ...
val submit: EventSrc[Int] = Event.source
```

In this implementation, the submit event source is created beforehand. While creating form, or any other element, attributes can be added. An additional attribute method listen binds event sources to certain properties, for example, onsubmit. This API makes the other trade-off, it keeps the element constructors referentially transparent by making the creation of event sources impure.

**Modeling a Variable Amount of Inputs**

An additional problem is how to model a variable amount of HTML inputs. Let’s look at an extension of the example in fig. 1 so that entries can also be removed by clicking on them. In the case of the event sources design we use a more advanced version of listen:

```scala
val delete: EventSrc[Int] = Event.source
def template(id: Int, entry: Entry): Html = 
  div(onsubmit.listen(delete, _ ⇒ id, ...))
val templatedEntries: DBehavior[List[Html]] = entries.map(_.zipWithIndex.map(template))
```

With the extended version of listen it is possible to modify the event value before invoking the event source. We use this to send an identification of the entry that should be deleted, something that is only possible if the event source is known before all entries. If we derive events from elements instead, we get the following code:

```scala
def template(entry: Entry): Html = div(...)
val templatedEntries: DBehavior[List[Html]] = entries.map(_.map { entryHtml ⇒ entryHtml.listen(_.onclick) })
```

In this case, template remains a simple implementation but entries no longer send to one specific event. Instead the input is modeled as a behavior of a list of events in inputEntries, which again, implies a need for higher-order FRP.

5. Pull-based Interfaces to the DOM

Previous sections focused on defining an FRP API that represents the DOM and its actions. In this section we look at how underlying FRP features can affect how close we can get to modeling all of the DOM’s features.

Applications written against the DOM are inherently event-driven, but elements in the DOM also have properties that can change over time, for example the value of a text input.

So far we saw two ways to retrieve DOM events, either by having a DSL that extracts them from an HTML element, or by plugging sources into elements upon creation. They are all modeled with FRP events, a push-based propagation primitive.

Existing FRP DOM libraries follow the designs we discussed in section 4 even when modeling DOM values, see for example: [7].
Let us take a look at how a date would be read for the to-do list application. We use the approach from section 4, Deriving Events from Elements to demonstrate the push-based read:

```scala
val dateInput : Html = input ( tpe := "text"
val date : CBehavior [String] = dateInput . read (_, .value)
```

The date is read in a push-based manner, the FRP DOM library subscribes to events on messageInput and propagates its changes. For each DOM value that is read, event handlers have to be registered and the actual state of the DOM is mimicked in the FRP system. Not only does this create overhead, it makes the correctness of a behavior rely on the ability to propagate all changes. Flapjax for example does not propagate changes that are made through Javascript property assignments (document.getElementById("field").value = "123"). FRP DOM libraries that follow this approach are difficult to use with existing Javascript libraries.

A common practice of web developers is to make use of pre-built widgets, for example, a date picker interface. For example, a date field is no longer just a text input, through added interface components a user can now enter a date by clicking through a calendar. The downside is that the underlying input text is no longer changed through user interaction but through the script’s additional interface. This causes values to change that do not have corresponding events. Push-based libraries are not notified automatically and are only correct if the developer adds additional event handler code.

5.1 DOM Properties as Continuous Behaviors

We propose to expose DOM properties in FRP through continuous behaviors, a pull-based primitive. Continuous behaviors do not suffer from the same issues as the push-based approach for property reads. When a value is required, for example, through the `sampledBy` or `snapshot` operations the property is read on-demand. In the case of the datepicker example, regardless of the `how` value changes, the correct value will be read on-demand from the original text input.

Much like in section 4, there seems to be a trade-off between two methods of linking continuous behaviors to the DOM DSL:

Deriving Behaviors from Elements. As an example we implement the to-do application's input fields:

```scala
val messageInput : Html = input ( tpe := "text"
val message : CBehaviorSrc [String] = CBehavior . src ("
val dateInput : Html = input ( tpe := "text"
val date : CBehaviorSrc [String] = CBehavior . src ("
```

In this implementation we create two simple inputs using the element constructors. Behaviors are extracted from the user interface elements through the `read` method. It accepts a function that defines which operation should be executed on the element when its value is queried. Just as before, the element constructor functions are no longer referentially transparent when such an API is added.

Creating Elements with Behavior Sources. The second example makes use of behavior sources:

```scala
val message : CBehaviorSrc [String] = CBehavior . src ("
val date : CBehaviorSrc [String] = CBehavior . src ("
```

In fig. 3, we bring all the pieces together and show the entire implementation. So far we have learned how to model the user interface, the input form and the corresponding behaviors and events. The only thing that we have not yet seen is line 7 to 15 in fig. 3, that is, the implementation of state. On line 7 we define `entry`, a
Flapjax first provided an FRP implementation for client-side web development in the form of a library and as a standalone web development framework [7]. Flapjax is a push-based DOM library which allows elements to contain behaviors. They do not have one explicit main value but allow the embedding of their HTML components, an approach that is similar to having multiple `DBehavior` main values.

A more recent web language that was based around FRP is Elm [1]; it is mainly focused on GUI development, but recent versions of the language no longer rely on FRP. By looking at how their interface library changed we can see that they tried both imperative solutions for the recursion problem that we discussed, first going with impure creation of input elements and then switching to an event source. Elm always had a push-based DOM API without abstractions to read properties from the DOM.

UI.Next [4] is an F# DOM library that focuses on providing a higher-order API for the DOM. They have a `DBehavior` main value and split the API in two layers, a dataflow layer and a presentation layer to tame the higher-order APIs.

In practice there are several other libraries that offer a reactive API for the DOM, for example React 7, a JavaScript library that uses a virtual dom approach to implement a component-based interface framework that provides declarative rendering. Its API is far from FRP. A more FRP-centered approach is Reflex-DOM 8, a Haskell library similar to UI.Next. It is based on the Reflex FRP library and allows (modified) higher-order primitives and follows an imperative monadic style of binding FRP to the DOM.

Acknowledgments

Bob Reynders holds an SB fellowship of the Research Foundation - Flanders (FWO). Dominique Devriere holds a postdoctoral fellowship of the Research Foundation - Flanders (FWO). This research was partially funded by the SBO project TEARLESS.

References


7https://facebook.github.io/react/
8https://github.com/reflex-frp/reflex-dom