Functional Reactive Programming

Maximilian Krome
Email: maximilian.krome@student.uni-luebeck.de

***Abstract*—FRP (Functional reactive programming) is an approach to use functional programming paradigms on processing dynamic signals. Those signals can either be continuous, so called "Behaviors" or discrete referred to as "Events". The inherited declarative aspect of FRP enables the programmer to model the solution as opposed to model the way to obtain it via imperative programming.**

FRP allows the programmer to write solutions that abstract away from platform and hardware specifics. Due to the nature of functional programming less code is needed resulting in less errors and increased readability. While being advertised as more intuitive, it has not overtaken classic imperative frameworks or programming languages.

FRP is used for GUI (Graphical User Interface) development. Although not limited to this field the seminar will focus on this aspect

I. Benefits of Functional Programming

In this section I will briefly cover some key benefits of functional programming over imperative programming that explain why FRP was developed in the first place. It is mainly based on John Backus' statements on functional programming [1].

A. Problems with Imperative Programming

Back in the beginnings of computer science Von Neumann founded the architecture of computers that is still in use today. It consists of a central bus that connects the CPU and memory, as well as the periphery. Imperative languages where designed by computer engineers in order to control this structure. They went from pure Machine code to Assembly to the modern programming languages we all know and use. Along the way they achieved some abstraction from the underlying hardware but still stuck close to the inherited Von Neumann architecture. Every command consists of some assignment of some computation. The assignment part is limited to only one memory unit at once and so are the commands. This splits the solution for any given problem in so many small steps that the original intent is most likely no longer visible. Of course more steps mean more code and more code inevitably leads to more mistakes and consequently more development costs. A further problem is the reliance on states in order to control the program flow. One word instructions prevent from doing otherwise since you need some way to take previous computation in account. This leads to the requirement of complex frameworks in order to control such state computations and everything that is designed on top of that framework is of limited expressiveness, because expressions and statements are strictly separated in these languages (again the one word at a time limit is at fault). For Example imagine your classic if, while and for statements in your imperative language of choice. Now try build your own statement of them. All of these problems came because languages where not build on a solid mathematic foundation but on hardware requirements. Therefore along these problems comes another one: It is very difficult to reason about the possible outcome of programs written in such languages in a scientific/mathematic way.

B. Solutions of Functional Programming

The pioneers of functional programming where aware of these issues and solved them developing, guess what, functional programming languages. Instead of looking at the given hardware and thinking of what to do with it, they designed languages looking at Math and thinking of implementing its properties. Inspired by Mathematics, functional programming languages come with a complete formal description of their syntax and semantics. Those descriptions can then be utilized for formal proofs of certain properties. And like in Math you describe objects rather than telling the computer how to get them. A very common example of functional programming is the faculty function:

```
fac :: (Int n) => n -> n
fac 0 = 1
fac n = n * factorial (n - 1)
```

Although it is entirely possible to write this similarly in, for example, Java, this is pretty much the mathematical definition written down. There is no fat to it. Code in functional programming languages comes close to specifying the problem itself. There is another very common example that embodies the idea behind functional programming very well:

```
qsort :: (Ord a) => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort less ++ [x] ++ qsort
    more
where less = filter (<x)  xs
more = filter (>=x) xs
```

The above example is an implementation of the quick sort algorithm in Haskell. It illustrates nicely how elegant and concise functional programming can be, there is no need to fully understand it. [1] The code does not contain any state transitions and the core idea of quick sort is immediately visible. Interestingly this little example already follows every possible paradigm of functional programming:

[1] If you really want to this may help: The first line is for type definition, not really interesting and can be ignored. ++ concatenates two lists; a:b pushes element a on list b; where allows you to define helper functions; in Haskell you do not put parenthesis around arguments. Haskell uses *pattern matching*, line three basically says: Do stuff when the input is a list containing at least one element and I want to refer to the first element as x while I will call the rest of the list xs. Goto the Haskell Wiki for a detailed explanation.

1) Referential Transparency: At no point it is valid to reassign variables. For example x = x + 1 would be invalid.
2) Purity: Functions do not have side effects and functions will always produce the same result when given the same values again.
3) First Class Functions or Higher Order Functions: Functions may appear anywhere every other object could do. More specific: They can be parameters or return values of other functions. For example the filter function takes a function that returns a boolean value for a single argument. With this property comes *partial application* and *currying*. Every function does not evaluate all its arguments at once but instead evaluates one argument, then returns a function that evaluates the remaining arguments the same way. For example 1 + 2 would first become function 1 + applied to 2. This is also exemplified above where the $<$ is only given one argument consequently returning another function.
4) Recursion: Since you are not allowed to reassign variables the standard for and while loop are useless leaving you with no alternatives but recursion. You can see it in the third line of example code.

How does all this knowledge help in regard to the original topic? The underlying tone is that every effort is taken in order to shun out those state computations. However you still need them in order to interact with the real world. A program may be supposed to act differently upon input if button x was or was not pressed beforehand, even though that is a violation of the purity paradigm. There are a variety of solutions for this problem:

1) Not making the functional programming language pure, meaning it allows for side effects and mutable values. Obviously this pretty much leaves only the First Class Function paradigm unharmed and is a major setback. It is also noted that many imperative languages support functional programming to some extend (Java and C sharp for example). Due to their imperative heritage they could never be purely functional without being a totally different language.
2) Using a special type system for mutable values. The prime example here is Haskell and its Monads. I wont go into a detailed explanation of Monads, but they are generally considered one of the hardest aspects of Haskell and functional programming all together. [2]
3) Always pass a state object as parameter around

Functional reactive programming takes the next step.

II. The Beginnings: Fran

A. Introduction

The idea of FRP was founded by Paul Hudak and Conal Elliott in an award winning paper published at the International Conference on Functional Programming 1997 [3]. They developed a set of functions and data types revolving around signals, specifically continuous behaviors and discrete events. Both of which are implemented as first class values, functions and signals can be combined with each other and themselves at will. Fran itself is not a real programming language but instead embedded in Haskell and running on Hugs (Haskell User Gofer System, basically a Haskell interpreter), thus they use Haskell syntax. They give a descent motivation for their work as they describe the properties of Fran, more precise Modeling. In the sense of functional programming modeling can be viewed as what you want to animate and presentation as how you want to do it. Ideally you do not have to bother with the later so you can focus on the former and that is what Fran wants to deliver. Similar to later papers on the matter of FRP, formal semantics of included functions and data types are presented. These contain the following:

1) Time: Is given as a basic behavior
2) Time transformation: the timeTransform function takes two behaviors as parameters, of which the second one is an alternation of the time signal. It returns the first behavior at the time value that the second behavior currently holds. timeTransform b (time/2) is behavior b slowed down by factor $1/2$.
3) Lifting: The liftn function takes a function with n parameters and n further behaviors. It can be thought of as a mapping function that takes the current values of each signal, feeds them to the function and returns the resulting values as another behavior. Lifting is so important and common, that it often happens implicit (like in the example above with time)
4) Integration: Works on numeric signals, takes a starting time and integrates the signal from that point on.
5) Reaction: Until takes a behavior and an event of a behavior as parameter. It returns a behavior that is either the one first given or the one contained in the event if the event happened.

Here a couple of basic examples:

```
colorCycle t0 = red 'untilB' lbp t0 *=>
        \t1 -> green 'untilB' lbp t1 *=>
        \t2 -> colorCycle t2
```

This behavior switches from red to green [2] every time the left mouse button is pressed lbp. lbp is parameterized with the time after which it should start producing values, like many other behaviors. The backslash initiates a lambda abstraction or anonymous function definition ($\lambda parameterName.term$ equals \ parameterName $->$ term). The $*=>$ is used to map events. [3]

```
followMouseRate im t0 = move offset im
 where  offset = integral rate t0
        rate = mouse t0 .-. pos
        pos = origin2 .+^ offset
```

followMouseRate takes an image and lets it follow the current mouse position at speed that is dependent on their current distance. Remind yourself of the equation for the relationship between velocity and distance: $s = \int vdt$. Here v is dependent on the distance between the mouse position and the current

[2] red and green are not just values in this context but behaviors that continuously carry the same value

[3] t2 $->$ colorCycle t2 could also be written as just colorCycle, because Haskell supports function currying

position of the picture. Since the mouse position is a 2 dimensional vector some vector specific operators are used: .−. creates a vector from to points, .+^ adds a point and a vector resulting in another point. offset, rate and pos are defined mutually recursive. It may seem odd at first, if these were normal functions they would call each other infinitely, never terminating nor returning a result. However since they are all behaviors things are a little different. Each of them map the current value of some other signal to their own current value. They do not care if the value they mapped originally changes under their feet as a result of their doing.

B. Behaviors

Behaviors could be implemented as follows:

```
data Behavior a = Behavior (Time -> a)
```

data is the Haskell key word for the definition of a constructor. The "a" on the left side is a type parameter similar to Generics in Java or Templates in C++. On the right side the actual parameters for the constructor are written, in this case just a function that takes time and returns a value of type "a". Unfortunately this does not work efficiently. When sampling a behavior that is dependent on some events it would be necessary to go back and look at every occurrences of these events every time sampling. Therefore behaviors could be defined like this:

```
data Behavior a = Behavior (Time -> (a,
    Behavior a))
```

When sampling this one receives the value and a new behavior. The new behavior is a (simplified) version of the existing one that starts out at the sampling time. This way there is no need to to redo the same work. The catch is that one must not sample with times earlier than with the time he sampled last. In fact behaviors are even more complicated:

```
data Behavior a = Behavior (Time -> (a,
    Behavior a)) (lvl Time -> (lvl a,
    Behavior a))
```

The first part is like the previous definition, the second is similar to that but operates on Intervals lvl of time and value. The purpose of this is explained below.

C. Predicate

One interesting problem is the translation of behaviors into events. Fran offers the predicate function which takes a boolean behavior and a time while returning an event that evaluates to () (unit type) when the behavior first becomes true after the given time. This bears a large problem: Boolean behaviors may evaluate to true for an infinitely small time to instantaneously

```
predicate (time * exp (4 * time) ==* 10)
    0
```

At this point it is no longer possible to get a correct program execution via sampling of signals. Symbolic solving of equations is not an option either since it only works on simple problems. Therefore *interval analysis* was introduced. Remember the second part of the final behavior definition? Well, it contains a function that returns an interval of values the behavior assumes in the given time frame. These functions are called *inclusion functions*. [4] Booleans can be considered ordered like False < True, so the Intervals [False, False], [False, True] and [True, True] are possible results of an inclusion function of a boolean behavior. For the predicate function this means the event time can be approximated: If the result of the inclusion function is [False, False] the event does not take place in the given timespan, if it is [True, True] it takes place immediately and [False, True] we cut the time interval in two halves upon which we call the inclusion function and check the results recursively. Of course it is not necessary nor possible to shrink the interval infinitely, instead one proceeds this formula until sufficient accuracy is achieved.

D. about Haskell as a host language

The type system of Haskell allows for relatively easy development of embedded languages. Haskell itself is, along other things, a purely functional programming language and passes its properties down to whatever language you build of it. Haskell is a lazy programming language, which means it does not compute its functions when called but when the return value is needed. This allows for infinite data structures, which in turn allow for very intuitive problem solutions. For example imagine you wanted every item of a list bound to its index you can call:

```
zip [1..] yourList
```

zip takes two lists and returns a list of tuples, each tuple containing corresponding elements of the lists. The length of the shortest list delimits the length of the resulting list. Therefore even if [1..] goes to infinity the program terminates, because it only builds the list as long as required by the second argument. [5] Generally it is nice to spare unnecessary computation, unfortunately there is a major drawback to it: The Haskell interpreter can pile up a very large stack of computations (space leaks) which then have to be executed all at once (time leaks). This greatly hinders responsiveness and consequently real time systems as well as GUIs. However you might have noticed that infinite data types are essential to the definition of behaviors (read above) and consequently the design of Fran would not have been possible the same way without laziness.

III. Real Time FRP

A. Introduction

RT-FRP [5] is an alternation that is designed to suit real time applications. Programmers in this field must be able to predict how much time and space any given computation may take (and it should not be much regardless.) Unfortunately that is impossible, it is well known that there is no program that

[4] How these functions do that is not explained in the paper regarding Fran but is the topic of another [4]

[5] If the second list is infinite the program will of course not terminate

can tell in limited time whether any given program terminates, let alone how much resources it uses. The only way around this is to limit the expressiveness of the language in which the program is written that you want to analyze.

RT-FRP is not designed to be used as language you write programs in, instead the main goal for it was to be an intermediate language. Classic FRP should be compiled in RT-FRP which would then be either compiled in C or interpreted.

Classic FRP has the full power of the lambda calculus with first class signals, so things like: behaviors of behaviors, recursive behaviors, function defining behaviors and vice versa are entirely possible and so are programs which do not terminate. RT-FRP has taken it upon itself to cut away expressiveness in such a way that you still end up with an useful language that meets real time constraints.

A few simple functions are provided in order to operate on signals:

1) ext lifts an expression to a signal. It can be seen as lift0 .
2) let signal x = s1 in s2 allows to access the current value of the first signal in an ext term forming the second signal. Example: let signal x = time in ext (x − 1) is the current time delayed by one second.
3) delay v s delays a signal by one tick. It displays v in the first tick.
4) s1 switch on x = ev in s2 switches from s1 to s2 whenever event ev occurs. Starts out as s1.

It has three main properties that guarantee its real time capabilities: It is statically typed and type preserving meaning runtime errors are impossible, costs for each execution step are bounded to a constant as well as the space for a running program and terms can not grow unbounded.

A strong type system is used to enforce these properties. In the end every well typed program does not get stuck in execution.

B. Syntax

RT-FRP has a very simple grammar describing its syntax that strictly separates values, terms and signals:

$$\begin{aligned} e ::= & x|c|()|(e_1,\ e_2)|e_\perp|\ \perp\ | \\ & \lambda x.e|e_1\ e_2|fix\ x.e \\ v ::= & c|()|(v_1,\ v_2)|v_\perp|\ \perp\ |\lambda x.e \\ s, ev ::= & input|time|ext\ e|delay\ v\ s| \\ & let\ signal\ x\ =\ s_1\ in\ s_2| \\ & s_1\ switch\ on\ x\ =\ ev\ in\ s \end{aligned}$$

e, v, s, ev refer to functional terms, values, signals and events. It is just the basic lambda calculus with some extensions for signal processing. The only function that can bridge the gap between functional terms and signals is ext, so the expressiveness is limited at this bottleneck.

C. Signals

Since signals are not first class, the things mentioned about classic FRP are not possible or very limited. It uses a simple isomorphism between behaviors and events:

```
Event a = Behaviour (Maybe a)
```

Maybe a is a Haskell data type which either evaluates to Nothing or Just a [6]. So if an event is happening the Maybe contains something and otherwise not. This trick enabled the developers of RT-FRP to combine behaviors and events into one data type. [7]

D. Recursion

Recursive signals are prohibited by the grammar of RT-FRP but still needed in some places, therefore let signal bindings

```
let signal x = s1 in s2
```

are allowed to be recursive to achieve the same effect. However, this is still too general since it can produce a faulty program even though it is well-typed:

```
let signal x = ext x in ext x
```

In order to limit this operations like ext are only allowed on a limited subset of expressions. It is possible to statically test if they fulfill this requirement.

E. Compiling FRP into RT-FRP

RT-FRP does not feature a lift function like Fran does. However it is possible to build this function for an arbitrary amount of parameters. lift1 e s ≡

```
let signal x = s in ext (e x)
```

Other common FRP constructions can be translated int RT-FRP as well.

F. Hybrid Automaton

HA is mix of continuous and discrete components. You can picture it as a graph with nodes that contain ”continuous” computations similar to what FRP does with behaviors. These nodes are then connected via edges which are labeled with a condition upon which they are traversed. Only one node is active at a time.

HA is a very important subject in the field of embedded systems. The most common example is a thermostat which in one state heats the room up until a threshold is reached then switches to another state where it lets the room cool down until the next threshold is reached when it switches back. Similar to RT-FRP safety and reliability are primary concerns (imagine the thermostat in a nuclear reactor or something). In order to be useful for embedded systems RT-FRP needs to be at least equally expressive as hybrid automatons are.

Nodes can be realized as signals but what about the transitions? Although s1 switch on x = ev in x2 is an option, it does not scale well with increasingly complex transition rules. Therefore RT-FRP provides parameterized signals with let continue {k_j x_j = s_j} in s and s until {ev_j => k_j} can be used as jump conditions for changing states.

[6] Similar to the Nullable class in Java

[7] Just a is written as $a_\perp$, Nothing is written as $\perp$

IV. Elm

A. Introduction

Elm [6], [7] is a language implementation of Functional Reactive Programming. It is neither the only one nor the first, but it is relatively well documented in a number of research papers which served as sources for this seminar. It is compiled to an intermediate language (like Java does with its Bytecode) and from thereon out to JavaScript. The development of web application is this languages main purpose, hence the JavaScript which runs in pretty much any browser nowadays. In contrast to the previous two languages Elm is designed with focus on practical use and not academics. It does not add more power to FRP in any way, there is no problem that cannot be solved with classic FRP but with Elm, though the Elm version may perform better. Elm provides a few simple functions in order to process signals:

1) lift . Self explanatory at this point
2) async. Marks code that can be executed independently from the rest
3) foldp It takes the current value of signal s and the accumulator a and feeds them to the function f. The result then replaces the accumulator. foldp f a s itself evaluates to a signal that contains all accumulator values.

That is it. No switch or until constructs this time. Elm is distinct from other implementations of FRP because it tackles two issues that most of them have: global delays and needless recomputation.

B. Syntax

$$
\begin{aligned}
e ::= & ()|n|x|\lambda x : \eta.e|e_1\ e_2|e_1 \oplus e_2| \\
& if\ e_1\ e_2\ e_3|let\ x\ =\ e_1\ in\ e_2|i| \\
& lift_n\ e\ e_1\ ,\ldots\ e_n|foldp\ e_1\ e_2\ e_3| \\
& async\ e \\
\tau ::= & unit|int|\tau \to \tau' \\
\sigma ::= & signal\tau|\tau \to \sigma|\sigma \to \sigma' \\
\eta ::= & \tau|\sigma
\end{aligned}
$$

τ, σ and η stand for simple types, signal types and types. Note that it prohibits signals of signals per definition: Only signals of simple types are allowed. In contrast to the grammar for RT-FRP functional terms or expressions of signals are allowed.

C. Signals

Contrary to the previous languages Elm does not feature continuous behaviors. Only a signal type is featured which is some mixture of behaviors and events: It always contains a value (like behaviors) and that value changes discretely. When it changes an event gets triggered which then causes computation of program. Other FRP versions may struggle on behaviors as they try to compute the program as often as possible in order to approximate continuation. As a result a great deal of needles recomputation is sorted out. Signals of signals are not allowed in order to ensure efficiency. There is a simple explanation why these signals are so problematic. It revolves around the function foldp f a s: Imagine you have a signal of signals of integers. You want to form a signal of all sums of the integer signals. It could go like this:

```
lift (foldp (+) 0) signalOfSignals
```

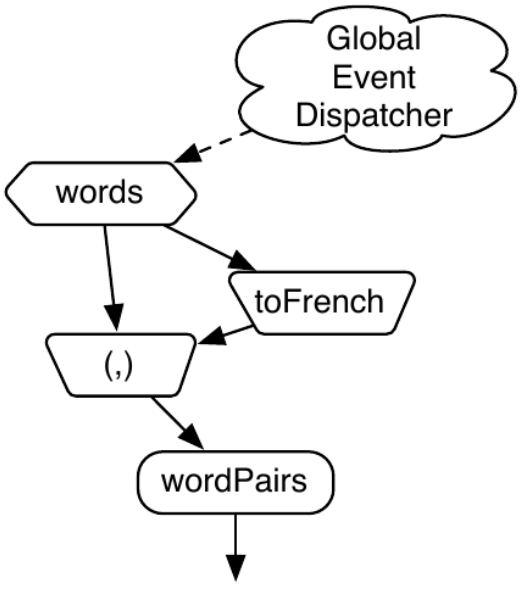
Now the problem is how should the foldp be evaluated? There are two options and both are flawed:

1) Every time the signal of signals produces a new signal, that signal gets evaluated in its entirety. This implies that every value any signal ever produced has to be saved. Consequently memory usage would grow proportionally to program execution time, ultimately exceeding all bounds. If we consider signals as sequences the example above would behave as follows: Input: < <1,2,3,4,...>, <5,6,7,8,...>, ...> Output: <<1,3,6,10,...>, <5,11,18,26,...>, ...>
2) Every new signal only gets evaluated from the point on it was produced. This could then lead to expressions that are equally defined yet evaluate differently, which is also undesirable. Assuming the signal changes at the second value, the same input as above would evaluate to something like: <<1,3>, <7, 15,...>, ...>.

Therefore signals of signals have been prohibited.

D. Graph representation

An Elm program can be considered as an acyclic graph. Its nodes are always signals, either input signals or compositions of signals. Input signals (like mouse button presses for example) do not receive any edges.



Whenever an event occurs at any input signal, *all* signals produce a value that trickles down the graph.[8] All signals which values have not changed produce the previous value

[8]Systems like this are called push based systems. They compute when input "happens", as opposed to pull based systems which compute when output is demanded

and mark it as unchanged. Nodes do not recompute input that has not changed, instead they just return their previous output, also marked as unchanged. This is one form of *memoization* and reduces needless recomputation. Since the graph is acyclic events can be *pipelined*. It is unnecessary to wait for an event to make its way through the whole program, if it finished the first stage the next event is ready to go.

E. Automaton

Automatons are a construct that allow changing single nodes in the program graph (read above). Remember the hybrid automaton section? This is basically another approach on that. Automatons are defined like this (again Haskell syntax):

```
data Automaton a b = Step (a -> (
    Automaton a b, b))
```

This seems similar to the definition of behaviors in Fran, as it follows along the same idea. Feed a value to an automaton and you get another automaton and a resulting value. Using it on signals:

```
step : a -> Automaton a b -> (Automaton a
    b, b)
step input (Step f) = f input

run : Automaton a b -> b -> Signal a ->
    Signal b
run automaton base inputs =
        let step' input (Step f, _) = f
            input
        in lift snd (foldp step' (
            automaton, base) inputs)
```

The step just applies the function that is in Step to the input. run takes an automaton, a base value and a signal. Every time the signal produces a value the automaton is applied to it. The resulting automaton replaces the current one while the resulting values form a signal that defines run.

F. Asynchronous Functional Reactive Programming

Global delays are not a problem limited to FRP, since they happen when languages demand that all events must be treated in the same order they happened and evaluation of an event is only allowed when the evaluation of the previous event is finished. Therefore a large computation can delay minor subsequent tasks that are completely unrelated to it. Elm provides the programmer with the async keyword to mark code that is independent and can be executed asynchronous from the rest.

G. "A Farewell to FRP"

Coincidentally the title of this subsection is the same as that of an article on the Website of Elm. The author Evan Czaplicki is also author of two papers on FRP: "Asynchronous Functional Reactive Programming for GUIs" and "Elm: Concurrent FRP for Functional GUIs" which both served as sources for this seminar. How come so? Well, it turns out the FRP parts of Elm where the hardest parts to learn and since Elm is all about easiness they were cut out and have been "[...] replaced with something simpler and nicer". Furthermore it is not even considered a big deal since most existing Elm code does not use signals anyways. The parts that use them can be replaced by a new system which revolves around subscriptions and web sockets.

V. Conclusion

Fran has initiated the development of FRP. RT-FRP catered it to the needs of real time applications. Elm used in order to create GUIs. They prove that the ideas behind FRP are suitable for a variety applications. They all treat behaviors and events differently in order to succeed: Fran has both in the form you would expect, RT-FRP has only behaviors and Elms signals are more like events. All of them work and allow the programmer to benefit from every aspect of functional programming when working with signals. While FRP has not overtaken classic imperative programming languages and frameworks, some ideas like events can be found in pretty much every GUI framework. They are even part of the core syntax of C#. So ideas of FRP stay prevalent, even though FRP itself is not so popular and Elm shifts away from it.

References

[1] J. Backus, "Can programming be liberated from the von neumann style?: A functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359579

[2] M. Lipovača.

[3] C. Elliott and P. Hudak, "Functional reactive animation," in *International Conference on Functional Programming*, June 1997, pp. 163–173.

[4] J. M. Synder.

[5] Z. Wan, W. Taha, and P. Hudak, "Real-time FRP," in *International Conference on Functional Programming (ICFP'01)*, 2001.

[6] E. Czaplicki, Ph.D. dissertation.

[7] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for GUIs," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, Jun. 2013, pp. 411–422.