Reactive Interfaces: Combining Events and Expressing Signals

Ragnar Mogk
Technische Universität Darmstadt
mogk@cs.tu-darmstadt.de

ABSTRACT

There are numerous attempts to design interfaces and abstractions for event handling in reactive applications. A common solution is to provide a set of combinator functions to manipulate and compose event streams, a technique originating from functional programming. Rich combinator libraries are essentially a domain specific language for common event operations. Another kind of interface for reactive abstractions are signal expressions, a syntactical construct to combine and transform signals – values that change over time. Each existing library has developed its own set of interfaces for reactive programming, often influenced by the interface of earlier libraries, and the actual need to support specific case studies.

However, no systematization of the developed interfaces exists. In this paper, we analyze the different interfaces provided by reactive languages. We group operations into several categories which relate to usages of reactive programming libraries, discuss how the choice of features influences the expressiveness of the respective library, and which implications interfaces have on the language implementation.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Data-flow languages

General Terms

Languages, Design

Keywords

reactive programming, event-driven programming, programming interfaces

1. INTRODUCTION

Reactive applications update their internal state and produce results in reaction to external events. This behavior addresses the requirements of a wide range of domains, such as user interfaces, Web servers, and sensor applications [22].

Traditionally, the main design issue when implementing reactive applications has been to decouple the code that detects or triggers the events, from the code that handles the events (i.e., to implement the semantic model of an application independently from the user interface). The Observer pattern [9] already achieves decoupling, but simple observers lack desirable features like composing reactions [7], so that complex reactions can be build from simpler ones. The syntax used for composition should also make the semantics and the control flow of the composition obvious and not hide them behind boilerplate code or scatter them around different places of the code – a problem caused by inversion of control [10]. Finally, it is desirable to integrate well with existing features of the host language, so programmers can use familiar tools as expected [23].

Today a wide range of libraries for reactive programming exist, which provide the features mentioned above, but have a different focus, expose custom interfaces and adopt different abstractions [24]. Examples of such libraries include FrTime [3], Fran [7], Reactive Extensions [16], Flapjax [19], and REScala [23]. In summary, programmers exposed to a variety of different interfaces even though the target problem is often very similar. Contextually, there is little or no discussion in the research community regarding the implications of these differences, and the relative pros and cons, resulting in a lack of guidelines that may ameliorate design choices for reactive programming interfaces.

In this paper, we analyze the interfaces for reactive programming, introduce the commonalities among the available abstractions and interfaces, and discuss examples uses of interfaces in a broad range of reactive applications, including the problems stated above. We also consider runtime requirements of certain features.

In summary this paper makes the following contributions:

- We provide an overview of the base abstractions used by reactive programming libraries.
- We describe the most common interfaces used by reactive programming libraries.
- We discuss the usage of different kinds of interfaces as well as their weaknesses.

The paper is structured as follows. Section 2 surveys common interfaces for reactive programming. Section 3 discusses the implications of certain design choices. Section 4 summarizes and provides recommendations for reactive programming libraries. Section 5 presents related work, and Section 6 concludes.

2. INTERFACE TAXONOMY

Existing libraries for reactive programming differ in the interfaces they provide. In this section, we identify three types of interfaces common to reactive programming libraries, (1) combinator libraries for observable event streams, (2) combinator libraries for time-changing values (i.e., signals or behaviors) and (3) signal

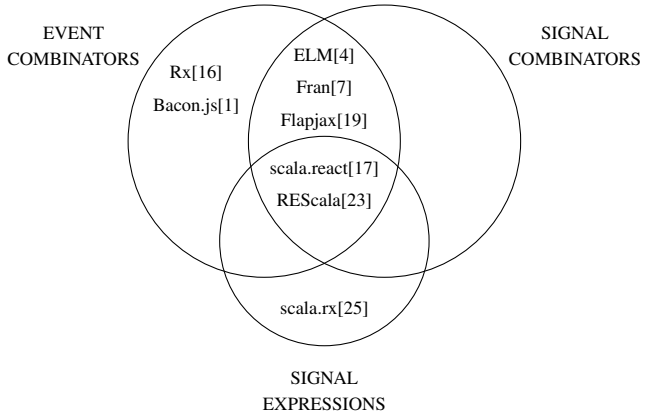

Figure 1: Reactive languages and interface abstractions.

expressions – a construct to allow derivation of new signals from arbitrary expressions.

Figure 1 shows how existing reactive programming libraries position in this design space, and Figure 1 shows an overview of the operations we present in this section.

Events	Both	Signals
Transform	Fold	Combine
Filter	Change	Flatten
Split	Read	
Merge		
Select		
Group		

Table 1: Operations on events and signals.

2.1 Combinator Libraries for Events

An event (a.k.a. event stream or observable) is an abstraction for a discrete-time occurrence – occurrences optionally carry a value. Events are common in reactive programming and most libraries dedicate many combinators to transform and compose events. For each library in the design space in Figure 1, we first provide an example for the use of combinator libraries for events, then illustrate a categorization of the operations which combinator libraries can support.

2.1.1 Example: Request Processing

The example in Figure 2 demonstrates the use of an event combinator library. A source event carrying external requests is tested for valid authentication and processed into a result and send back to the requester. We use Scala-like syntax in code examples – they can be easily transferred to any concrete library as the concepts are similar for all combinator libraries.

Common operations include transforming events by filtering occurrences with a predicate, such as the authorization status of the request (Line 2), and transforming values based on a given function, which compute part of the result from the request (Line 4). More complex operations include merging multiple events. This functionality is demonstrated in Line 5 of the example, where another part of the result is looked up, and both parts are combined by the `zipWith` combinator using the `combineParts` function. It is also possible to react to an event with a side effect as an exit point of the reactive library connecting to external interfaces, the `sendResult` function in this case (Line 7).

```
1 def source: Event[Request]
2 val filtered = source.filter(isAuthorized)
3 val result = filtered.
4   map(calculateResultPart).
5   zipWith(filtered.map(lookupResultPart))
6          (combineParts)
7 result.observe(sendResult)
```

Figure 2: Use of combinator libraries for events.

The example also illustrates the use of combinator libraries as a domain specific language, where the combinators in Lines 3-6 are written in the order reflecting the control flow of the event transformation pipeline – a common syntax in reactive programming.

2.1.2 Operation Categories

Combinator libraries often provide a rich set of combinators, implementing many operations. We group available operations into categories, with each category providing distinct functionality for events.

The simplest operations on events take a single input event and produce a single derived event as an output. The categories differ in how the occurrences of the resulting event are derived from the occurrences of the input event.

Transform Applies a function to the value of the input event occurrence, and returns the result of the function application as the new value of the derived event occurrence. The derived event always occurs if the input event occurs. The `map` combinator is an example of a transformation operation.

Filter Tests the value of the input event occurrence. The derived event only occurs if the test is successful. The value of the derived occurrence is the same as the input occurrence. Figure 2 shows the `filter` combinator.

Split Produces multiple event occurrences from a single one. The derived event occurs multiple times in reaction to a single occurrence of the input event. For example, the result in Figure 2 (Line 5) may be large, and split into multiple smaller parts, each as its own occurrence of the derived event.

Most frameworks also support operations on multiple events. We distinguish two distinct categories of combining events, and a single category to produce multiple result events.

Merge Combines the values of multiple occurring input events into a single result value. The derived event occurs only if all input events occur and has the combined result as the value. Merge generalizes Transform to multiple input events.

Select Selects one occurrences from multiple input events and return it as the occurrence of the derived event. Select generalizes Filter to multiple input events.

Group Take a single input event and group occurrences into one of multiple result events. An example would be an event producing key value pairs, split into one derived event for each key. The derived events occur with the value, when the input event occurs with the corresponding key

Operations on events can belong to multiple categories, a single operation could transform and filter an event at the same time. Combined operations can be more natural to use and offer implementation benefits. For example, when combining *Select* and *Split*, multiple occurrences of the input events can be selected and then split into the single derived event, without requiring any intermediate data structures – improving code clarity and reducing overhead.

2.2 Combinator Libraries for Signals

Events alone are unsatisfactory when the application deals with changing state. Consider an application which uses a temperature sensor. Events are well suited to observe each temperature change but if the current temperature value needs to be accessible anytime, the application must store the temperature and update it upon event occurrence. Time-changing values – such as the temperature – are common in reactive systems and often connected with event occurrences. Many reactive programming libraries thus include a dedicated abstraction called *signal* (or behavior) to model time changing values in addition to events.

We first provide an example how to create and use signals and then take a systematic look at operations available in libraries with signals and events.

2.2.1 Example: Request Statistics

A first way to create signals is to observe properties about the history of an event. Figure 3 shows an example where the number of incoming request events is counted and stored as a signal (Line 8). Every result is collected into a list of results (Line 9). Such derived signals are automatically updated every time the input event occurs.

```
8 val requestCount: Signal[Int] = source.count()
9 val allResults: Signal[List[Result]] = result.list()
```

Figure 3: Transforming events to signals.

A snapshot of the current value of a signal can be accessed at any time, which allows to use the value in a non reactive context. However, a typical task is to derive a new value based on the snapshot, and it is often desirable that the derived value updates as a reaction to changes of the input signal. For this purpose, a new set of combinators can be used to derive new signals from existing ones. Figure 4 shows an example that derives the number of results by counting the length of the result list (Line 10). Deriving a value from multiple input signals is done with the lift combinator (Line 12), which takes any number of signals and a function with the same number of arguments, and derives a new signal by applying the function to the values of the input signals whenever one of them changes. In this case, we test whether the difference between the number of request and sent results exceed a threshold. The derived signal then contains the boolean result of the test.

```
10 val resultCount = allResults.map(_.length)
11 def threshold: Int
12 val exceedsThreshold = lift(requestCount, resultCount){
13   (reqC, resC) => reqC − resC > threshold
14 }
```

Figure 4: Combinator libraries for signals.

2.2.2 Features for Signals and Events

In contrast to events, signals on their own do not support a diverse set of operations. Instead, signals are used to derive new signals from multiple input signals, similar to the *Merge* operation on events.

Combine Derives a new signal based on the value of one or multiple input signals.

Combining multiple signals is simpler than combining events, as signals always have a valid value, where events may or may not occur. Both `map` and `lift` in Figure 4 are examples for signal combination.

There are three new operations to support combinations and conversion between events and signals.

Fold Derives a signal from an event. The value of the result signal changes on every event occurrences, by applying a function to the old value of the signal and the value of the event occurrence. The operations folds the sequence of event occurrence with the signal value as an accumulator. The `count` and `list` combinators in Figure 3 are both examples of specific fold operations.

Change Derives an event from a signal. The result event occurs every time the signal changes to a different value. The value of the occurrence contains the old and the new value of the signal.

Read Any operation on events can, in addition to its normal inputs, also read the current values of any number of signals. Examples for possible combination are transforming or filtering events based on a signal value, or taking snapshots of signal values when an event occurs.

2.3 Signal Expressions

Signal expressions are a syntactical construct for the *Combine* operation on signals, the only operation to derive signals from other signals stated in Section 2.2.2. While multiple signals can be combined by lifting functions, the combination of signals is often very specific – the lifted function is defined inline and only used once for the purpose of the lifting. We first show an example of how signal expressions better integrate into the host language, and then show two advanced features which integrate very well with signal expression syntax.

Figure 5 shows the previous threshold example, but uses signal expressions instead of function lifting. A signal expression is an expression which accesses the values of other signals (in this case using function call syntax), and is wrapped in a `Signal` block to indicate the scope. The resulting value of the expression is the value of the newly generated signal, automatically updated upon changes in the input signals.

```
12 val exceedsThreshold = Signal {
13   requestCount() − resultCount() > threshold
14 }
```

Figure 5: Basic signal expressions.

A major advantage of signal expressions is that they work with all expressions of the host language including expressions involving complex control flow. Figure 6 shows an example of a signal expression testing a signal as part of a condition. Notably, the value of `allResults` is never accessed as long as `exceedsThreshold()` is `false`. The signals accessed inside a signal expression can be dynamically detected by the runtime, and used to determine when the derived signal needs to change.

```
15 val resultsIfThreshold = Signal {
16   if (exceedsThreshold()) allResults()
17   else Nil
18 }
```

Figure 6: Signal expressions with conditional access.

Dynamic detection allows the use of higher order signals – signals with other signals as part of their values – inside of signal expressions. Figure 7 shows an example with a higher order signal (Line 2). The list of clients is wrapped in a signal, because

clients may be added or removed. The name of each client is represented as a signal (Line 1), as it may also change. Making the name of a client at a specific position available as a signal requires nothing more than undoing the three nesting levels inside a signal expression. The derived signal depends on three signals, the list of `clients`, the `activeID`, and the name of the accessed client. Note that the accessed client can not be known statically, so requires dynamic dependency discovery.

```
1 trait Client { val name: Signal[String] }
2 val clients: Signal[List[Client]]
3 val activeID: Signal[Int]
4 val activeClientName: Signal[String] = Signal {
5   clients().at(activeID()).name()
6 }
```

Figure 7: Accessing nested signals.

The use of higher order signals is enabled by the *Flatten* category of operations, which are naturally supported by signal expressions

Flatten A signal can have other signals as part of its value. Inside of a signal expression, the value of nested signals can be accessed, resulting in a dynamically selected dependency. The resulting signal depends both on the inner as well as the outer signal.

Note that combinators libraries can also support a flatten operation, and thus require dynamic dependency discovery. However, flattening is very natural with signal expressions and needs to be supported to make many syntactically valid expressions work.

Signal expressions also support all available operations that only involve signals, so libraries which only support signals can opt to have signal expressions as their only interface.

3. DISCUSSION

This section discusses advantages and disadvantages of the interfaces presented in Section 2, and analyzes how specific design choices of interfaces can influence the implementation of the runtime. We first state the advantages of combinator libraries and why they are often used for reactive programming libraries, and explain the problem of combinator libraries becoming too large.

We then discuss signal expressions in detail, including why they can solve the problem of large interfaces only for signals, but not for events. We also detail design considerations for the syntax of signal expressions. Lastly, we discuss selected semantics and performance differences of reactive programming runtimes, which depend on the chosen interfaces.

3.1 Advantages of Combinator Libraries

Nearly every library for reactive programming provides combinators for events and signals. We find three major reasons for the success of combinator libraries. (1) The concise and declarative expression of chains of operations on reactive values, (2) the availability of combinator syntax in most general purpose languages, and (3) the safe extensibility of libraries by adding new combinators.

3.1.1 Data Pipelines

Combinator libraries are well suited to describe data pipelines, where the data from a source is transformed multiple times by a sequence of combinators, and then processed by a final sink operation as demonstrated in Figure 2. The source code reflects the pipeline operations in the same order as they are semantically executed, making it easy for a human reader to follow the program control flow.

Data pipelines are common not only in reactive programming, but also in traditional areas of programming languages, such as collections, optional values, and future values. Rx describes observables (a variant of events) as a combination of a collection and a future [21], filling a gap in the standard library of many programming languages.

3.1.2 Availability

Combinator libraries are supported by most general purpose programming languages independent of the underlying programming paradigm. While combinator libraries originate in functional programming, they only rely on the availability of higher order functions.

Combinator libraries are actually a reason to include higher order functions in a language. An example is Java and the Java stream API [12], which leverages lambda expressions introduced to the language at the same time [11] to enable a declarative and concise syntax for stream pipelines, similar to the reactive pipelines of the discussed in the previous section.

Availability in many languages allows for different reactive programming libraries to share concepts and interfaces between libraries for different languages. A good example is the Rx family of libraries [16], which has libraries available for many languages, all sharing similar combinators. This also allows programmers to learn libraries for reactive programming more quickly.

3.1.3 Extensibility

Combinator libraries are extensible for the author of the library. New operations can be supported by adding more combinators to a basic abstraction, without interfering with existing operations, or requiring the programmer to learn new syntactic constructs. On the other hand, it is also possible for a library to only implement a subset of the operations for reactive programming listed in Section 2, because combinators are independent of each other.

Extensibility is important for reactive programming libraries, as many aspects of libraries are still under active research, and new interactions with the library are required for external integration.

Adding more combinators also has disadvantages as discussed in the next section.

3.2 Drawbacks of Combinator Libraries

Combinator libraries can grow quickly in the number of available combinators, leading to large library surfaces complicating the use of the library.

Each operation in a combinator library has to be expressed only by using proper combinators. When no suitable combinator for a given use case is available, there are essentially three options to achieve the desired effect: (1) using multiple combinators, (2) using a generic combinator and (3) falling back to a direct manipulation of the library abstractions. We will first take a look at multiple and generic combinators and then at direct manipulations.

3.2.1 Multiple or Generic Combinators

Figure 8 shows an example for implementing the count combinator in Line 1 with multiple combinators (Line 2) or a generic combinator (Line 3).

These solutions however imply loss in clarity, as programmers need to figure out the interactions of multiple combinators, or what exactly the specific instance of the generic combinator computes.

A second drawback is potential loss of performance when using

```
source.count()
source.list().map(_.length)
source.fold(0)((a, _) => a + 1)
```

Figure 8: Alternatives for `count`

multiple combinators, as each operator adds another layer of indirection. Even without indirection, a specialized combinator can often improve performance over the generic version. For example, the `count` combinator can completely ignore the value of the event occurrence, while the `fold` combinator needs to unpack the value and pass it to the function, where the value is then ignored.

This performance burden can be significant when the desired operation is simple as in the example in Figure 8.

3.2.2 Direct Manipulation

Implementing the operation by directly manipulating the underlying abstraction is often the most complicated choice, but may be necessary if there is no suitable combinator.[1] However, implementing combinators directly requires knowledge about the internal invariants of the library, and is often only practical with modifications to the source code of the library, unless the library provides an interface for direct manipulations.[2] The concrete syntax for direct manipulation of the library abstractions varies from library to library, a possible interface using subclasses is sketched in Figure 9, where a count event is implemented which occurs with an increasing count every time the input event occurs. In the example it is assumed that events alway occur on a single thread and mutable state can be safely used as a reaction to occurrences.

```
class Count extends DerivedEvent {
  var counter = 0
  def onInputOccurence() = {
    counter += 1
    occurWithValue(counter)
  }
}
```

Figure 9: Sketch of a subclass based version of `count`

To avoid the aforementioned issues, libraries include a number of combinators to cover all expected use cases. As a result they expose very large interfaces, putting a burden on the user who has to learn many different combinators. While it is possible to use only the needed combinators without knowledge of the whole interface, finding the necessary combinators often involves navigating large parts of the library interface.

3.3 Signal Expressions

A significant advantage of signal expressions is to allow arbitrary combinations and transformations of signal values, as a signal expressions allows to use normal host language expressions without restrictions, and adds the possibility to access signal values. A signal always has a value, so combining two signals has the trivial semantics of using the current value of each signal.

This design does not require a rich set of combinators as operations on signals are expressed in familiar syntax. As a result, signal expressions do not suffer from the problem of large interfaces discussed previously.

When it comes to discrete change occurrences, signal expressions show a number of limitations. An event may or may not occur. Just combining and transforming the values of occurrences is not as expressive as it is for signals. For events, sometimes it is needed that all involved events occur (such as a `zip` operation), which could be handled with a corresponding event expression. Other operations require only some of the events to not occur (i.e., computing an average of just the occurring event values).

An event can be seen as a signal with an optional value, which allows the event to be embedded into a signal expression. However, the signal expression has to deal with these optional values, bloating the expression with additional statements, thus removing the advantages of signal expressions compared to combinator libraries.

3.3.1 Dependency Discovery Scope

Section 2.3 shows signal expressions dynamically detect the set of accessed signals. This approach implies that dependencies are not only detected inside the static scope of the signal expression, but also in the dynamic scope of the signal expression. Figure 10 shows an example where a signal is accessed outside the static scope of a signal expression (Line 2), but the execution happens inside the dynamic scope (Line 3). Dynamic scoping has the risk of adding unwanted dependencies to the signal expression, so it is desirable to disallow access to signals outside of the static scope of signal expressions.

```
def source: Signal[Int]
def add(i: Int): String = source() + i
val results = Signal { add(5) }
```

Figure 10: Signals in the dynamic scope of a signal expression.

Note that confining signal access does not limit the possibility to access signals dynamically. Signals can still be selected dynamically (even outside the static scope of the signal expression), only the final access to the value of the signal has to happen inside the static scope.

3.3.2 Implicit Value Access

Signal expressions require to explicitly extract the values of the signals used inside the expression. Bainomugisha et al. [2] refer to this approach as manual lifting in contrast to implicit lifting where the value of a signal is extracted without additional syntax. Figure 11 shows the threshold example with implicit lifting.

```
13 val exceedsThreshold = Signal {
14   requestCount − resultCount > threshold
15 }
16 val text = Signal { exceedsThreshold.toString() }
```

Figure 11: Implicit signal value access.

Implicit lifting reduces syntactic noise and makes the content of the signal expression look just like a normal expression. However, this solution exhibits two major disadvantages.

First, similar to the issue discussed in Section 3.3.1, it becomes harder for a reader to spot the use of signals. This problem is demonstrated in Figure 11 Line 14, where it is impossible to know which of the three used values are signals without looking at the type of the used variables.

Second, the code can become ambiguous when a value supports the same operation as a signal. For example, the receiver of the call to `toString` in Line 16 is ambiguous, it could refer to the signal

[1] RxJava has no combinator to provide a running count which occurs every time the count increases, it only provides a combinator which produces a count after the counted event signals that it will not occur any more.

[2] REScala requires the use of private methods to access event and signal values, which are not normally available to user code, so direct manipulation of library abstractions is not possible.

abstraction itself, or the contained value – the programmer has no way to specify the desired behavior.

3.4 Synchronicity

The operations on events and signals described in Section 2 are supported by multiple frameworks but their underlying abstractions have different semantics, resulting in similar combinators from different libraries having subtle semantic differences.

Consider the merging operator `zipWith` in figure 2. The basic idea is to take two event occurrences and combine them into a single occurrence. However, different libraries have different semantics for what it means for two events to occur simultaneously. One possibility of `zipWith` semantics is to accept only events occurring at the same time, that is, if the event occurrence is derived, possibly transitively, from one occurrence of a common shared input event. Another option is to wait without blocking until both events have at least one occurrence to trigger the composed event, in this case, the occurrences do not need to share a common transitive input.

The underlying difference between the two interpretations of `zipWith` lies in the way reactions are interpreted. In the first case, the reactions to the external change are considered to happen synchronous, all at the same time, and only events occurring together at the same time can be combined. The semantics of any operation combining multiple events are fixed, resulting in more predictable reactions to changes.

In the second case, there are several asynchronous reactions to external changes, where each event is handled individually occurring at independent times depending on the underlying implementation. The timing semantics of event combination are defined individually by each operation, leading to more flexible semantics at the cost of predictability.

3.5 Runtime Performance Implications

Signal expression require dynamic dependencies to correctly support language features like conditionals. Dynamic dependencies require a runtime which can discover the dependencies used during the evaluation of a signal expression. Also, the runtime needs the ability to dynamically add and remove dependencies to and from a derived signal.

This dynamic discovery process can add significant overhead to signals with cheap computations. If the runtime manages a graph of the dependencies between signals, then each potentially dynamic access needs to be intercepted and the graph changed accordingly. For push based change propagation, a dynamically detected dependency can also cause unnecessary computations, if the changes are not yet propagated to the new dependency. On the other hand, for expensive computations dropping a dependency while it is not needed can have significant reductions in computation cost.

Note that combinator libraries do not suffer from these issues because they do not provide dynamic discovery of dependencies for most combinators. Fortunately, a runtime can support dynamic discovery and only pay the price when it is used by signal expressions, but not when combinators with static dependencies are used.

In principle, static code analysis could be used to detect the use of signals inside of signal expressions to remove the overhead of dynamic checks in case all accessed signals are known in advance. To our knowledge static analysis has not yet been explored in the context of signal expressions.

4. SUMMARY AND OUTLOOK

Combinator libraries provide a concise way to define a transformation pipeline for events. However, to support all desired functionalities, interfaces become large and harder to learn and understand. Signal expression solve the problem of large interfaces for signals, but are unsatisfactory for all the operations required by events and signals. Especially for pipelines of events, combinators provide a well-suited syntax, which is also shared by other libraries for similar data pipelines (e.g., collection or stream processing libraries).

Reactive programming libraries thus can and should support both combinators and signal expressions, but keep the set of combinators focused on the core tasks of the library. Still, it needs to be easy for the user to learn available combinators. Libraries should avoid having combinators with surprising semantics – combinators should feel similar to another. A user should be able to predict the semantics of an unknown combinator based on known combinators. If different semantics are necessary it may be better to introduce a completely separate interface, for example by introducing a new abstraction with its own set of combinators.

To facilitate interactions not provided by the included combinators, the library should provide a safe way to directly manipulate the basic abstractions. We believe that there is still research needed on how a library can achieve exposing a flexible enough interface, without giving the user the ability to break properties guaranteed and managed by the library. In the context of direct manipulations, a more complex version of signal expressions, which also allows to specify the different operations on events may be suitable, as such an interface might be expressive enough while still requiring less boilerplate code than implementing a library extension (c.f., Figure 9).

5. RELATED WORK

Bainomugisha et al. [2] provide a comprehensive survey on reactive programming languages, with a focus on implementation model and features supported by the individual implementations.

Fran [7] is one of the early libraries to support both events and signals (or behaviors, as signals are called in Fran). Fran is implemented in Haskell, and combinators are used as the library interface. The target domain of Fran are animations, so signal values are sampled continuously to produce an image output. The representation of a signal in Fran reflects the sampling behavior, and conceptually corresponds to a function from the current time to the signal value, but needs to be more complex because of technical details [6]. In the context of pure functional programming, Arrowized FRP [20] provides a different interface, where functions to transform signals are manipulated instead of the signals themselves. Arrowized FRP allows for efficient implementations of reactive frameworks in languages where referential transparency is required.

The abstractions adopted in Fran influenced many newer approaches to reactive programming in other languages. FrTime [3] is one influenced library implementing reactive programming in Scheme. FrTime uses Schemes macro system to provide automatic lifting of functions, which makes the use of signals transparent to the programmer. Flapjax [19] implements reactive programming for Javascript, eliminating the need for callbacks in common event driven Javascript applications. Flapjax provides a templating mechanism to embed signals as part of the displayed HTML. Another approach influenced by functional reactive programming is Reactive Extensions [16], a library to improve integration of event handling in mainstream applications, which uses the convenient syntax combinators provide for event pipelines.

More recent advances in reactive programming often focus on the language runtime. Elliott et al. [8] propose a push based propagation of updates for Fran to reduce unnecessary recomputations of

signals compared to the continuous sampling. ELM [4] prohibits the use of dynamic signals to allow pipelined execution of a push based update propagation. The execution strategy of ELM allows the integration of non blocking execution of long running tasks into signals. Scala.React [17] integrates with thread pools of external libraries, and also introduces a domain specific language to combine temporal sequences of event occurrences. Reactive programming has been also extended to distributed applications [5, 18]. In the distributed context, a major issue is to provide a an algorithm for propagating updates without the need for central coordination.

Another recent line of research uses advanced type systems to guarantee runtime properties such as bounded-space execution [15], absence of space and time leaks [14] and liveness [13].

6. CONCLUSION

In this paper, we analyzed existing designs for reactive programming languages. Combinator libraries are a common interface choice. We argued about their flexibility and support for diverse operations but also showed the drawback of a large and possibly hard to learn surface area of available combinators. We discussed signal expressions, a syntactic improvement to make combinations and transformations of signals more natural. We show the use cases of signal expressions – combination of many signal values used almost identical to regular values in expressions of the host language.

Finally we recommended that libraries should focus on a small set of important combinators, and provide an extension interface to allow for arbitrary interactions for when the provided combinators are insufficient.

7. ACKNOWLEDGEMENTS

This work has been funded by the LOEWE initiative (Hessen, Germany) within the NICER project, and partially supported by the European Research Council, grant No. 321217.

8. REFERENCES

[1] BaconJS website. https://baconjs.github.io/.

[2] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013.

[3] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *ESOP*, pages 294–308, 2006.

[4] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. PLDI '13, pages 411–422. ACM, 2013.

[5] J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini. Distributed REScala: An update algorithm for distributed reactive programming. OOPSLA '14, pages 361–376, New York, NY, USA, 2014. ACM.

[6] C. Elliott. Functional implementations of continuous modeled animation. PLILP '98/ALP '98, pages 284–299. Springer-Verlag, 1998.

[7] C. Elliott and P. Hudak. Functional reactive animation. ICFP '97, pages 263–273. ACM, 1997.

[8] C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 25–36. ACM, 2009.

[9] Gamma, Helm, Johnson, and Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2000.

[10] P. Haller and M. Odersky. Event-based programming without inversion of control. volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer Berlin Heidelberg, 2006.

[11] JSR 335: Lambda expressions for the Java programming language. https://jcp.org/en/jsr/detail?id=335.

[12] Java stream API documentation. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

[13] A. Jeffrey. Functional reactive programming with liveness guarantees. ICFP '13, pages 233–244. ACM, 2013.

[14] N. R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. ICFP '13, pages 221–232. ACM, 2013.

[15] N. R. Krishnaswami, N. Benton, and J. Hoffmann. Higher-order functional reactive programming in bounded space. POPL '12, pages 45–58, 2012.

[16] J. Liberty and P. Betts. *Programming Reactive Extensions and LINQ*. Apress, Berkely, CA, USA, 1st edition, 2011.

[17] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[18] A. Margara and G. Salvaneschi. We have a DREAM: Distributed reactive programming with consistency guarantees. DEBS '14, pages 142–153. ACM, 2014.

[19] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. OOPSLA '09, pages 1–20.

[20] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. Haskell '02, pages 51–64. ACM, 2002.

[21] Rx introduction. http://reactivex.io/intro.html.

[22] G. Salvaneschi, P. Eugster, and M. Mezini. Programming with Implicit Flows. *Software, IEEE*, 31(5):52–59, Sept 2014.

[23] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. AOSD '14, 2014.

[24] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: an analysis and a research roadmap. AOSD '13, pages 37–48, 2013.

[25] Scala.rx Web site. https://github.com/lihaoyi/scala.rx.