Visualizing Reactive Execution History using Propagation Traces

In Progress Paper

Takumi Hikosaka
Ritsumeikan University
Japan
is0230si@ed.ritsumei.ac.jp

Tetsuo Kamina
Oita University
Japan
kamina@acm.org

Katsuhisa Maruyama
Ritsumeikan University
Japan
maru@cs.ritsumei.ac.jp

Abstract

Reactive programming is an emerging programming paradigm where reactive behavior in modern software that periodically responds to changes in surrounding environments is naturally and declaratively represented. One well-known technique for debugging reactive programs is time-traveling where we can pause the execution and rewind to any earlier point in the execution history. On the other hand, anomalies in a data stream often appear in the form where the stream sometimes conveys an error, which may repeat over time. To find such an error, an aid to overview the execution history might help. In this paper, we propose a tool that visualizes the execution history of a program written in a reactive programming language to help the programmer find when suspicious updates of reactive values occur in the history. We also propose a method to record the execution history that makes this visualization possible. Then, we provide a couple of research questions that will be answered in the future work.

ACM Reference format:
Takumi Hikosaka, Tetsuo Kamina, and Katsuhisa Maruyama. 2018. Visualizing Reactive Execution History using Propagation Traces. In *Proceedings of REBLS'18, Boston, United States, November 4th, 2018,* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

Reactive programming (RP) is a paradigm where change propagation between reactive values, each of which constitutes a data stream in that its value is periodically updated, is declaratively specified. There have been lots of programming languages that directly provide language features to support RP, and it has been shown that reactive behavior in modern software that periodically responds to changes in surrounding environments is naturally represented in such languages. Examples are reactive extensions (Rx) to lots of modern programming languages, functional reactive programming (FRP) languages [1, 2, 15] where we can declaratively specify networks of time-varying values (i.e., *signals*), and languages that integrate signals with event-based programming [4, 6, 12].

One well-known technique for debugging programs in RP is *time-traveling*. Elm provides a time-traveling debugger that enables programmers to pause the execution and rewind to any earlier point in the execution history [7]. Reactive Inspector [13], a debugger for REScala [12], also provides a similar mechanism that is combined with visualization of signal networks.

One limitation of such time-traveling debuggers is that the history of execution is perceived only through animations and thus we cannot grasp the whole execution history in a single display. For example, anomalies in a data stream often appear in the form where the stream sometimes conveys a wrong value. Thus, we first would like to overview what values the stream had conveyed in the execution, and then directly go back to the execution point when the wrong value was produced to inspect the reason for this anomaly by following the dependency graph of signal networks. However, using the time-traveling debuggers we need to carefully replay the execution to find the execution point where some suspicious update of reactive value occurs, which can be a time-consuming task.

This paper proposes a tool, which is an extension of Reactive Inspector, that visualizes the execution history of a program written in an RP language to help the programmer find when suspicious updates of reactive values occur in the history. One issue in implementing this tool is how to represent *one execution step*. In Reactive Inspector, this step corresponds to one update or propagation of a signal value; i.e., while the propagations in a single connected signal network *theoretically* occur at the same time, the debugger shows every step in the propagations that are sorted and sequentially executed. However, from the viewpoint of the execution history, the visualizer should show that those

REBLS'18, Boston, United States

DOI: 10.1145/nnnnnnn.nnnnnnn

propagations occur at the "same time," which is more natural w.r.t. the semantics of RP languages [4]. Thus, in our visualizer, one execution step consists of propagations within the same connected signal network. We also consider that our tool enhances program comprehension in RP languages, as it shows the logical timing of reactive execution that is coherent with the RP language semantics.

To determine the set of propagations that occur in the same time, additional analysis to log the execution history is necessary. Assuming a glitch-free RP language, our tool maintains a logical time whose update is synchronized with update propagation in a single connected signal network. Our tool records this logical time for each signal update; thus, our tool can determine which updates occur at the same time.

Finally, we raise a couple of research questions, which will be answered in the future work, to study whether the proposed tool is really effective for RP debugging and program comprehension.

The contributions of this paper are summarized as follows:

- The design of time-traveling visualizer, which is coherent with the RP language semantics w.r.t. the update propagation timing, where we can overview the history of update propagations in the signal network to help the programmer find errors that sometimes occur in the history.
- The method to record the execution history that can be applied to our visualization tool.
- The design of our future research in the form of research questions.

2 Motivation

2.1 RP: A Brief Overview

RP has been proposed to address the limitations of procedural languages, which heavily depend on callbacks to represent reactivity, and such callbacks make programs difficult to understand, analyze, and achieve separation of concerns because of the inversion of control triggered by them. Two language constructs, signals and events, play important roles in RP.

A signal directly represents a time-varying value, i.e., a value that is a function of time. For example, the following code fragment shows motor control software in REScala [12], where the power of the motor is determined using a function, namely, `f`, according to the current sensor value.

```
val powerDifference = Signal{ f(sensorValue()) }
```

This declares a signal (time-varying value) `powerDifference` that depends on `sensorValue`, which is also a signal. By accessing `sensorValue`, we can obtain the value of the sensor *at this time*, and by accessing `powerDifference`, the power difference *at this time* is obtained accordingly. This means that every update in `sensorValue` is automatically propagated to `powerDifference`. Thus, these signals declaratively represent the functional dependency between the sensor value and the motor.

The other mechanism is an event, which is useful when signals interact with existing imperative programs. For example, the motor will be accompanied by the API for controlling it, which is implemented as a legacy library where the power of the motor is set by an imperative operation. In such a case, we can represent an update of `powerDifference` as an event, and register an event handler that propagates this update to the library function, namely, `adjustMotor` that changes the power of the motor. REScala provides several built-in functions that return an event. For example, `changed` is a function that returns an event that is fired everytime the value of the receiver signal is updated. The following code fragment registers an event handler that is called everytime the value of `powerDifference` is changed to call `adjustMotor` with current value of `powerDifference`.

```
powerDifference.changed +=
    { (e) -> adjustMotor(e) }
```

2.2 Debugging for RP

RP mechanisms are now well-studied, and known to be useful for a variety of application domains including, e.g., Web applications [6] and embedded systems [14]. As RP languages abstract the underlying computation details in a quite different way from the imperative languages, we should consider a different way of tool support for software development using RP [13]. In this paper, we focus on a debugging support for RP.

To inspect what is going on in a program written in an RP language, there have been two useful ways of tool support that do not appear in the traditional breakpoint debugging for imperative languages: (1) visualization of signal networks and (2) time-traveling. In RP, computations occur in the form of update propagation throughout the signal networks that represent the dependency between signals. Reactive Inspector [13], a debugger for REScala, revealed that visualizing how such networks are constructed and evolved, and showing which value is conveyed in a particular part of propagation are quite useful for debugging for RP. These features are achieved by setting a breakpoint on each creation and update of a signal.

Time-traveling enables programmers to pause the execution and rewind to any earlier point in the execution history, which was realized in the Elm debugger [7]. Reactive Inspector also provides a similar feature that enables back-in-time debugging, i.e., changes in the signal network over time can be inspected by navigating the execution history back and forth.

Figure 1 shows the visualization of signal dependency in Reactive Inspector. Each rounded rectangle represents a reactive node in the network such as a signal or an event handler. This is a snapshot of a particular execution point, and each

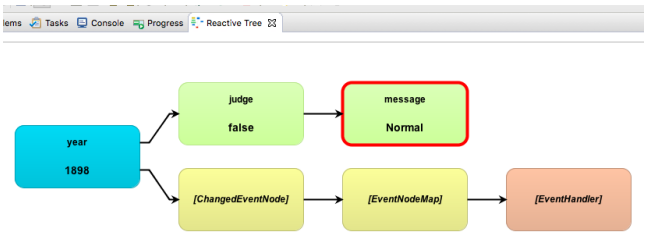

Figure 1. Visualizing a signal network

```
val year = Var(0)
val judge = Signal{
    year()%4==0 ||
    (year()%100!=0 && year()%400==0) }
val message = Signal{
    if (judge()) "Leap Year" else "Normal Year"}
year.changed += { _ => println(message.now) }
```

Figure 2. A bug in judging leap year.

signal contains the value at that execution point. We can observe how update propagation proceeds by replaying the execution in a stepwise or back-in-time manner. We can also inspect how this network is constructed and evolved.

2.3 Limitation in Existing Debugging

One limitation of time-traveling is that the execution history is perceived only through animations; it plays as a flipbook where each page is a snapshot of the signal network at a particular time. On the other hand, anomalies in a data stream, which constitutes values of a signal over time, often appear in the form where the stream *sometimes* conveys a wrong value. Thus, inspecting such values over time would be helpful to find the bug. However, in the time-traveling setting, we cannot overview the list of execution points in a single display.

To be precise, we use an example program that judges whether every each year that passes in a data stream in the order is leap year (Figure 2). This example is written in REScala, and both `judge` and `message` are signals that convey boolean and string values, respectively. The variable `year` is also a signal, which is special, as it does not depend on any other signals and its value is imperatively updated. In this paper, we call such a signal a *source* signal. The signal `year` conveys integer values, each of which is consumed by `judge` to evaluate whether current value of `year` is a leap year. This judgment is consumed by `message` to produce a string representation for that judgment that is finally displayed by the event handler registered to `year.changed`. This handler is called every time the value of `year` is updated.

This program is not correct, as it accidentally judges every multiplier of 100 as a leap year. This kind of failure might be overlooked, as it does not always produce a wrong value; at least 99 of 100 values in `message` are correct in this program. If we could overview the history of `message`, it would be much easier to find what is going on, because we can find the *pattern* of anomaly that is repeated over time.

3 Approach

To address this limitation, we propose a visualization of the execution history, where we can overview all the past computations in a particular signal network to help programmers find some anomaly patterns that are repeated over time. To this end, we extend Reactive Inspector, the aforementioned debugger for REScala, which already provides a mechanism to record the history of signal networks that can be a basis for our visualization tool.

3.1 Propagation Traces

To visualize the execution history, we need to preserve an execution trace [10], which is a sequence of *execution points*. To make our discussion precise, we need to define what is an execution point. For example, an execution point might be every step in the program execution. However, such a fine-grained execution point is not meaningful for our purpose, as we only focus on signal updates and their propagations. For example, to enable time-traveling, Reactive Inspector records every signal update, as well as every construction of signal network and every call of event handler. These records are necessary for back-in-time debugging and replay of the execution. In other words, Reactive Inspector defines an execution point as an update of a signal. So the question is whether this definition is suitable for our purpose.

To answer this question, we first need to see the fact that, when visualizing the execution history, we are concerned with the *logical time*. Actually, every connected update propagation triggered by an update of a source signal occurs in the logically *same time*. Even though each update of signal in the same connected network is performed in different real time, *semantically* there is no time progress. This semantics enhances the language understandability in particular for the RP languages that do not provide any explicit notion of time [4]. For example, when `year` in Figure 2 is updated, `judge` and `message` are simultaneously updated. Those updates are considered a single atomic operation, and there are no inconsistent situations where, e.g., `year` is updated while other connected signals remain in the old values.

Even though Reactive Inspector is useful to inspect the flow of propagations (e.g., inspecting the propagation order), to show the execution history, we require a more coarse-grained definition for an execution point. We define an execution point as a set of signal updates that are performed in the same logical time. We note that, to visualize a sequence of such execution points, we need to record additional information in the log that indicates the set of signal updates

that occur in the same time, which is explained in the next section.

3.2 Recording Propagations in the Same Time

To make it possible for our visualization tool to identify the set of signal updates that occur in the same time, the tool tags each signal update with the time when that update occurs. For example, consider the case where the existing tool can record the name of signal and the new value when that signal is updated, and the format of this record looks like: `signalName:newValue`. We extend this record to include the update time: `signalName:newValue, time`. Thus, the execution history of the example in Figure 2 looks like as follows:

```
1 year:1898, 20352
2 judge:false, 20352
3 message:Normal Year, 20352
4 ...
5 year:1899, 20362
6 judge:false, 20362
7 message:Normal Year, 20362
```

This log indicates that the first three lines (lines 1-3) and the last three lines (lines 5-7) occur in the same time.

It is obvious that the time in the records should not be the *current* time obtained by the environment. While update propagations in a connected signal network occur in *logically* the same time, these are sequentially performed in possibly different real time. Thus, we need a mechanism to maintain a logical time at runtime that is different from the real time.

This logical time is incremented everytime a new update of a connected signal network, i.e., an update of a source signal (`Var`), is detected. As REScala is a glitch free language, each update of logical time can be synchronized. In our tool, the value of this logical time remains in the same value until all propagations in the connected network end. Thus, when recording updates of signals, we can record the same logical time for the propagations in the same connected network.

We note that an invocation of an event handler is performed *after* the update of the corresponding signal. For example, the event handler for `year.changed` in Figure 2 is called after the update of `year` (and of course after the update of every connected signal), and thus `message` used in the event handler always conveys the value after the update. This means that the call of event handler is performed in different logical time from the update of the corresponding signal.

3.3 Visualizing the History

The analyzed propagation trace is visualized as shown in Figure 3. This diagram shows a history of the signal network in Figure 2 and its associated event handler. We assume that a single network is selected for visualization. Horizontal arrows indicate the progress of the logical time. Dashed vertical arrows indicate the update propagations, and solid vertical arrows indicate the event handler calls. Each oval indicates a value of the signal at a particular logical time. Signal updates tagged by the same logical time are placed in the same vertical line; thus, it is easy to see the propagations that occur in each execution point. It is also easy to see that each event handler call is triggered after the update propagations.

A suttle issue arises when we consider evolution of the signal network. In REScala, a signal network is evolving dynamically. For example, in Figure 2, the source signal `year` is created first, and then `judge` is created and connected with `year`. These creations and connection occur in *different* logical time; e.g., `year` may be updated before `judge` is connected with that. Our visualization tool should reflect such timings in the evolution of the signal network.

Figure 4 shows how our tool visualizes evolution of the signal network. Each creation of signal that is created later is not placed directly below the formerly created signal but shifted to right, which means that, e.g., `judge` is created and connected with the network *after* the creation of `year`, and `message` is created and connected with the network *after* the creation of `judge`. In the log record, each creation of signal is assigned with the different logical time; thus we can easily implement this shifting. In this example, the source signal, `year`, is not updated during the evolution. Thus, initial values for each `judge` and `message` are calculated using the initial value of `year`.

Figure 5 shows the case where there is a branch in the signal network. In this case, each year in the Japanese Era is calculated for each year in the Christian Era. This diagram shows that there is a branch at the update of `year`, and to reflect the fact that this branch is updated at the same logical time with the update of `judge` and `message`, the sink of branch is located in the same vertical line with the sinks to `judge` and `message`.

Even though the aforementioned examples contain only a single signal network, we note that this mechanism can easily be extended to multiple signal networks, as our mechanism rely only on the logical time. One issue is how to filter irrelevant parts of the network from a large scale diagram. For this purpose, we are planning to develop an extension of the query mechanism provided by Reactive Inspector. We also note that our signal networks are limited to those represented in Reactive Inspector. For example, a nested network (i.e., a signal of a signal) is not visualized in our tool.

4 Discussion and Research Questions

Currently we are developing a visualizer of RP execution traces that is proposed in this paper. Our hypothesis is that this visualization enhances efficiency of finding bugs in the programs written in a RP language. By "efficiency" we mean

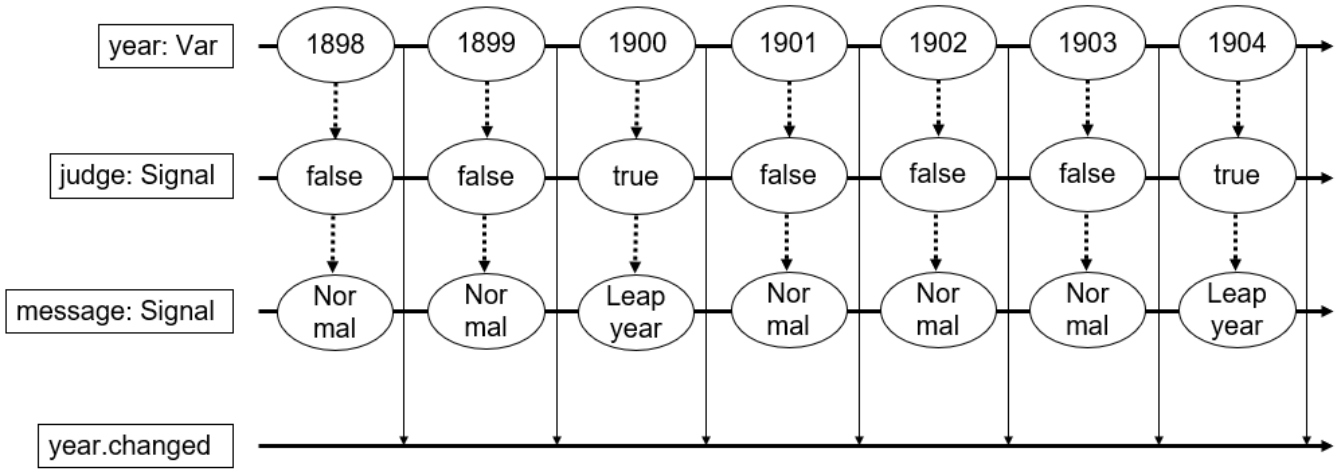

Figure 3. Visualization of the history of Figure 2

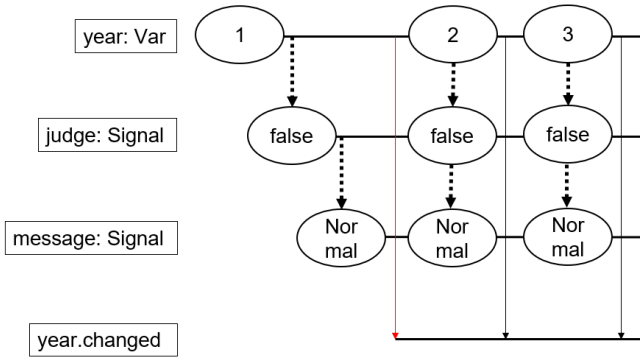

Figure 4. Visualizing evolution of the signal network

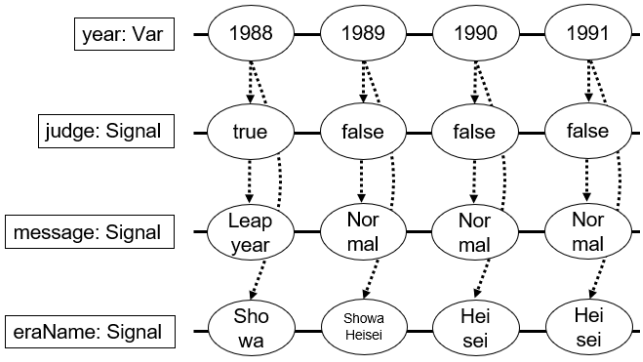

Figure 5. A branch in propagation

both cost efficiency and high bug detection rate. An advantage of our tool is that our tool shows each signal as a data stream while preserving the visibility of dependency between signals. By showing each signal as a data stream, we can easily see anomaly patterns that are repeated over the stream. For example, in Figure 3, we can see the pattern that every multiplier of 100 is accidentally judged as a leap year.

We also consider that this tool enhances program comprehension in RP languages. This tool visualizes the RP language semantics in that every update in the same connected signal network occurs in "the same time," and the handler is called *after* the update, i.e., a signal referred in the handler holds the value after the update. This might be tricky for novice programmers. For example, in Figure 2, a programmer who is used to imperative languages would consider that `judge` is updated after `year` is updated, and `message` is updated after `judge` is updated. This consideration is not a mistake, as it reflects the propagation ordering. However, this programmer might accidentally consider that the event handler for `year.changed` is called just after the update of `year` and before the update of `message`. Our visualization correctly shows the (logical) timing of propagations and event handler calls. Thus even a novice for RP can easily understand that `message` referred in the event handler holds the value after the update propagations.

However, these our prospections have not been validated yet. To validate them, we raise the following research questions whose answers will be provided in the future research.

RQ1. Is debugging of reactive applications with the proposed visualization easier than with debugging only with time-traveling?

RQ2. Does the visualization help novices to understand programs written in an RP language?

5 Related Work

While RP languages are now well-studied, how to support RP in the entire development process, including debugging, using a proper tool ecosystem remains as a research issue. Reactive Inspector [13], which we have introduced in this paper, is the debugger that visualizes update propagations and

the evolution of the signal networks. This also enables back-in-time debugging, similar to time-traveling, that enables the execution to rewind to any earlier execution point. Our tool is based on this debugger to make us possible to overview the execution history. Yampa Debugger [8] demonstrates particular challenges in testing and debugging in interactive applications can be eased in a pure FRP language. Its GUI tool visualizes an input stream and violations of assertions based on temporal logic. On the other hand, our tool puts more emphasis on how a particular signal is related with other signals, and how functional signal updates interact with side-effective event handlers. The similar illustration of timeline can also be found in Elmsvuur [3] that applies the timeline to show multi-tier communication.

Using traces to incorporate the notion of time to enable programmers navigate over the execution was first proposed in trace-based debugging [10]. Several trace-based debugger have been developed for OO languages [5, 9, 11]. These debuggers visualize global traces that consist of execution threads and method calls, as well as local traces for particular data or control flows, to facilitate navigation over execution. Based on RP, our tool puts more emphasis on showing logical timing for computations in RP such as signal update propagations and event handler calls.

6 Conclusions and Future Work

A tool that visualizes the execution history, which consists of signal update propagations and event handler calls, has been introduced. We designed a time-traveling visualizer where we can overview the history of update propagations in a connected signal network and its associated event handlers. We have argued that this tool makes debugging of reactive applications easier, as it helps us to find some anomaly pattern that repeated over time in the data stream. We have also argued that this tool enhances program comprehension in RP languages. Our prospections are, however, not validated in the paper.

As future research, we are planning to conduct some controlled experiments to answer the aforementioned research questions. In this experiments, we will measure both time and bug detection rate with and without using our tool. We will also provide several tasks to understand programs in a RP language to novice subjects and measure a percentage of correct answers with and without using our tool.

Acknowledgment

This work was sponsored by the Grant-in-Aid for Scientific Research (15H02685).

References

[1] Gregory H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language.* PhD thesis, Department of Computer Science, Brown University, 2008.

[2] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 263–273, 1997.

[3] Jeff Horemans and Bob Reynders. Elmsvuur: A multi-tier version of elm and its time-traveling debugger. In *TFP 2017*, volume 10788 of *LNCS*, pages 79–97, 2017.

[4] Tetsuo Kamina and Tomoyuki Aotani. Harmonizing signals and events with a lightweight extension to java. *The Art, Science, and Engineering of Programming*, 2(3), 2018.

[5] Bil Lewis. Debugging backwards in time. In *AADEBUG'03*, 2003.

[6] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA'09)*, pages 1–20, 2009.

[7] Laszlo Pandy. Bret Victor style reactive debugging. Elm Workshop, 2013.

[8] Ivan Perez and Henrik Nilsson. Testing and debugging functional reactive programming. *Proceedings of the ACM on Programming Languages*, 1, 2017.

[9] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *OOPSLA'07*, pages 535–552, 2007.

[10] Steven P. Reiss. Trace-based debugging. In *AADEBUG 1993*, volume 749 of *LNCS*, pages 305–314, 1993.

[11] Kouhei Sakurai and Hidehiko Masuhara. The omission finder for debugging what-should-have-happended bugs in object-oriented programs. In *SAC'15*, pages 1962–1969, 2015.

[12] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY'14)*, pages 25–36, 2014.

[13] Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In *ICSE'16*, pages 796–807, 2016.

[14] Kensuke Sawada and Takuo Watanabe. Emfrp: a functional reactive programming language for small-scale embedded systems. In *MODULARITY Companion*, pages 36–44, 2016.

[15] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *PADL 2002: Practical Aspects of Declarative Languages*, volume 2257 of *LNCS*, pages 155–172, 2002.