From Debugging Towards Live Tuning of Reactive Applications

Ragnar Mogk
Technische Universität Darmstadt
Germany

Pascal Weisenburger
Technische Universität Darmstadt
Germany

Julian Haas
Technische Universität Darmstadt
Germany

David Richter
Technische Universität Darmstadt
Germany

Guido Salvaneschi
Technische Universität Darmstadt
Germany

Mira Mezini
Technische Universität Darmstadt
Germany

ABSTRACT

Directly visualizing the effect of changes to applications improves developers productivity as they gain immediate insights on the resulting application behavior. Yet, immediate feedback requires a representation of the dataflow in the application to correctly propagate and apply the effect of the changes.

Reactive programming is a programming paradigm which directly expresses applications dataflow in a declarative way. Recently, researchers developed dedicated debugging techniques for reactive programming that use the dataflow graph to enable inspections and visualization.

In this paper, we adopt reactive debugging as a basis for live modifications and tuning of reactive applications. To this end we extend the debugger to allow live modifications to the dataflow graph in a structured manner to change the application behavior. We also suggest how developers of reactive applications can use such extension to enable flexible tuning of applications at run time. Our early experience shows that the combination of reactive programming and live programming allows modifications and tuning of applications while ensuring safety and consistency guarantees.

KEYWORDS

Reactive Programming, Debugging, Live Tuning, Live Programming

1 INTRODUCTION

Reactive programming uses high-level abstractions to directly express the dataflow of reactive applications – those applications which update their internal state based on external events and automatically propagate state changes. Reactive programming improves readability of complex dataflow and helps preventing bugs [21]. We base our work on reactive programming because it (a) enables the development of interactive applications in a natural style, (b) is well-suited for visualization, (c) is flexible enough to be adapted at run time by non-experts, and (d) gives the runtime enough information about the program to provide guarantees such as data safety, i.e., data is not lost or processed by an intermediate inconsistent version of the program when it is modified at run time.

Debuggers for reactive programming already allow inspection of a reactive application through its dataflow graph during execution [20]. Based on reactive debugging, we pave the way towards live programming. First, we extend an existing reactive debugger to allow run time modifications to the dataflow graph in a structured manner. We provide some of the benefits of live programming to application developers while ensuring that modifications always stay within the confinements defined by the application code, i.e., guarantees given by the type system still hold, and data dependencies in the application are kept consistent.

Second, we investigate how reactive applications may be inspected and modified not only by developers but also by users of the application. Our goal is to enable developers to make their applications tunable by domain experts using the application. To this end, we propose patterns and idioms as well as new language abstractions for specifying tuning points of the applications for domain experts and, at the same time, still provide basic safety and correctness guarantees of the application.

We assume three levels of expertise. First, the *language and tool authors* who design very general abstractions and mechanisms to implement arbitrary reactive applications. This first group, however, has no knowledge of the domain or requirements of the final application which is written by the second group – the *developers*. Developers use the abstractions and tools of the language to implement concrete application semantics. Developers determine concrete use cases for their application. They are responsible for ensuring that the application behaves as desired. We assume they are familiar with the abstractions provided by the language and are able to use them correctly. The third group, the *domain experts* of the application, have concrete tasks they want to solve with the application. Because tasks often vary slightly and those variations are not always predictable by developers, domain experts also need to vary the behavior of the application. However, because domain experts do not have prior knowledge of application development, it is more appropriate for them to change existing behavior in well-defined ways, such that the application solves their tasks, without the risk of breaking the application. We use the name *tuning* [15] to refer to this process in the rest of the paper.

The main contributions of this paper are:

- We outline how the technique of reactive change propagation along the dataflow graph corresponds to and can be used to apply changes at run time in a live environment.
- We extend an existing reactive debugger to allow live modifications.
- We present early work towards tunable reactive applications.

In the rest of this paper, we first give a background on reactive programming (Section 2) and the existing reactive debugger (Section 3). We enable live updates to the running application and explore new functionality for reactive debuggers to modify the state

LIVE'18, November 2018, Boston, Massachusetts, United States

of the dataflow graph during the application run time (Section 4). We then discuss how the modifications capabilities can be extended by developers to provide tunable applications to domain experts (Section 5). Finally we discuss other approaches and related work (Section 6).

Live Demo. As our contributions are best demonstrated live, in the workshop demo we will show the functionalities of our extension of the Reactive Inspector debugger (Section 3). We plan to demonstrate how the combination of reactive programming and live programming enables principled live modifications of reactive applications, as well as how such applications are tunable to achieve the desired functionalities driven by live feedback.

2 REACTIVE PROGRAMMING

Today's reactive programming is derived from functional reactive programming (FRP). FRP has been first applied to functional modeling of animations [13]. Early FRP systems compute individual frames of an animation given timestamp t of the frame. FRP languages provide composable operators for the computation of frames. In these systems, computation is *synchronous* and *pull-based*, i.e., when the frame at time t is requested, all operators are evaluated at that time and the resulting values are pulled into one final result.

Reactive programming has been introduced into a number of mainstream languages mostly targeting interactive applications in the GUI domain [10, 16]. In the interactive setting, pull-based evaluation of continuous time operators is found to be inefficient because all input events have to be stored and reprocessed every time a frame is requested. To solve this inefficiency, push-based evaluation is introduced [12]. Input events are eagerly pushed into the *dataflow graph* – the graph of connected operators. For each input, operators recompute and cache their current value. Evaluation semantics is still synchronous, i.e., all *reactions* (recomputations of operators) to an input event happen at the same time. The system produces the current state of each operator *now*, and no inspection of the history is possible.

The inability to inspect the history makes the application harder to understand because the current state depends on the order in which external events occurred and can no longer be expressed as a pure computation of time. We discuss how this problem is addressed in reactive debugging in Section 3.

2.1 REScala

We use REScala [19] for our examples. REScala adopts the synchronous, push-based propagation model, and supports two kinds of reactive abstractions, *events* and *signals* for discrete or continuous time-changing values, respectively. A signal always holds a value, and represents state in the program, e.g., the current text of an input field or the cached average of past sensor readings. An event only has a value when the event *fires*, and it represents actions that occur in the program, e.g., when a button is clicked or a message is received. The example in Figure 1 demonstrates a prototype application that visualizes sensor data in a dashboard. `Evt` (Line 2) serves us as an input from an external temperature sensor. The temperature event is processed by an operator pipeline (Lines 5, 7) filtering extreme temperatures (probably sensor faults) and aggregating the last 50 temperatures – a signal derived from

```
1  // input sensor for temperature
2  val temperature = Evt[Double]
3  // filter very high and low temperatures an outliers
4  val filtered =
5    temperature.filter(_ <= 100).filter(_  >= -40)
6  // build a history of up to 50 temperature values
7  val history: Signal[Seq[Double]] = filtered.last(50)
8  // an aggregation function that may change over time
9  val aggregation: Var[Seq[Double] => Double] =
       Var(average)
10 // apply the aggregation function to the temperatures
11 val aggregated = Signal {
12   val f: Seq[Double] => Double = aggregation.value
13   f(history.value)
14 }
15 // select which values to display in the dashboard
16 val onDisplay: Var[List[Signal[Seq[Double]]]] =
       Var(List(history, aggregated))
17 // assume a generic rendering function for values
18 def render = {
19   case d: Double => ...
20   case l: List[Double] => ...
21 }
22 // render selected signals on the dashboard
23 val dashboard = Signal {
24   for (item <- onDisplay.value)
25     yield render(item.value)
26 }
```

Figure 1: Dashboard application.

an event. Events and signals may be derived from other events and/or signals as long as no cyclic dependencies are formed. The aggregation function (Line 9) can change over time, i.e., we model it as a signal, which holds the current aggregation function for every point in time. More specifically, we model it using a `Var`, which is an input "signal" that can be imperatively set. The signal expression (Line 11) applies the (time-changing) aggregation function to the (time-changing) history of temperature events. Derived events or signals are automatically updated whenever an input changes. Inconsistent intermediate values based on a set of inputs which are only partially updated yet are prevented – a property called glitch freedom. The dashboard (Line 23) then renders all values that should be on display (Line 16). Because the list of values on display is a signal (a `Var` to be precise), the concrete displayed items may change, resulting in a dynamic dataflow graph. The REScala manual and the API documentation[1] provide further details about each REScala operator.

3 REACTIVE DEBUGGING

Debugging is the process of understanding the behavior of code to resolve bugs. Debuggers aid programmers at finding defects and help code comprehension by visualizing the application behavior

[1] www.rescala-lang.com/manual

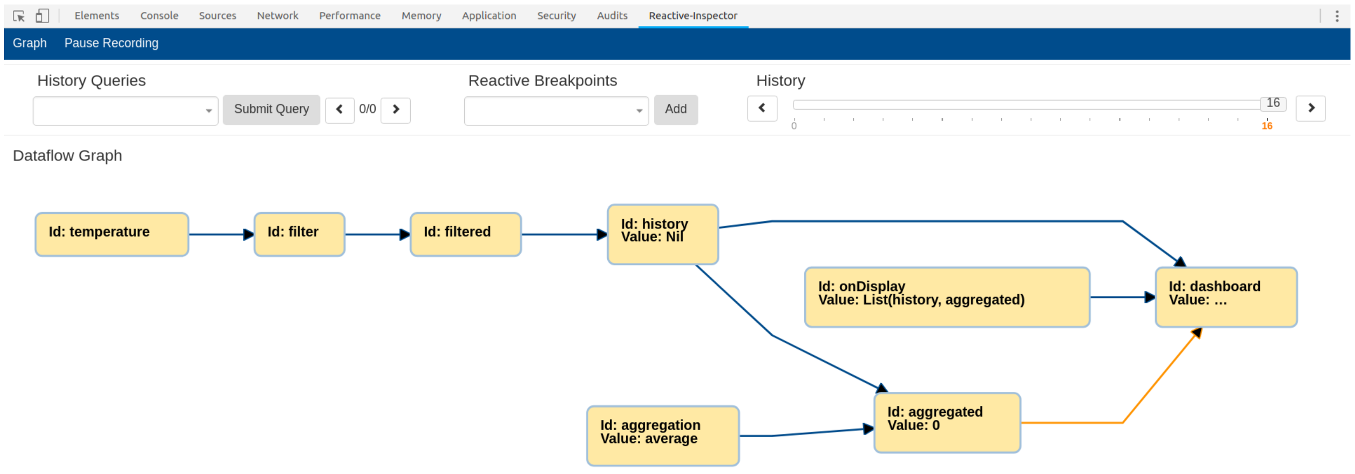

Figure 2: Reactive Inspector user interface.

during run time. Traditional debugging tools focus on the analysis of control flow in sequential programs. They provide means to interrupt the application at certain points (breakpoints) and continue by stepping through each instruction of the program. However, debugging based on control flow is unsuited for reactive programming languages, because the control flow view does not correspond to the dataflow of the reactive application. Therefore, developers often use "printf debugging" to trace the dataflow or develop their own ad-hoc visualizations of the dataflow graph [5].

Reactive debugging [20] is a debugging technique designed to fit the needs of reactive programming. Reactive Inspector, a tool that implements this technique, provides a set of features similar to those of traditional debuggers but adapts to a dataflow view, which is shown in Figure 2. Reactive Inspector directly visualizes the dataflow graph and enables navigation through the history of changes using history queries and a time slider. Reactive Inspector can also pause execution of the current application when selected nodes in the dataflow graph are modified, similar to breakpoints in sequential debuggers.

Visualization. Figure 2 shows the Google Chrome debugger extension of Reactive Inspector. Reactive Inspector displays the dataflow graph for the dashboard application code presented in Figure 1. The main view of the debugger shows the dataflow graph. Nodes in the dataflow graph are named after their corresponding variables in the source code. Data flows from left to right, beginning with the temperature event and ending at the dashboard signal. The view shows the dataflow graph directly after it has been created, so no temperatures have been recorded yet. The highlighted edge (orange) represents the latest modification to the graph – the connection between the aggregated value and the dashboard. When the graph changes dynamically during program execution, i.e., new nodes or new edges are added, those changes are reflected in the dataflow visualization.

History. History navigation is similar to stepping through a sequential program, but does not pause the execution of the application and works both forward and backwards. As described in Section 2, reactive applications only store the current state – it is normally impossible to inspect past states of the graph – hindering understanding of the past behavior of the application leading to the current state. Reactive Inspector solves this issue by storing every past state of the dataflow graph and providing users with a way to visualize the history of the graph. The history can be inspected by moving the history navigator slider shown in Figure 2, and points of special interest can be quickly found by issuing a history query. History interactions are only visualizations, the application is neither modified nor paused – new events are recorded in the future while inspecting the past.

Breakpoints. Similar to the breakpoints in traditional debugging tools, Reactive Inspector allows users to attach breakpoints to certain graph nodes which cause the program to halt when the corresponding node changes its value. Breakpoints may be conditional and only trigger on specific values passing through nodes. A breakpoint can also be set on the creation or deletion of nodes from the dataflow graph. When a reactive breakpoint is triggered, Reactive Inspector delegates to the native Chrome debugger to step through the sequential code inside of signal expressions.

4 FROM INSPECTING TO MODIFYING

The reactive paradigm is particularly promising for live programming because reactive applications are already designed to support dynamic changes – which enables interactively experimenting with different value in the graph. For example, the developer might wonder what would happen if a very high (and thus filtered) temperature was measured when the history already contained 50 elements. *Would this temperature still trigger the update of the history and remove one old element, even though it was filtered?*

Yet, despite the dataflow graph is designed for processing dynamic and arbitrary input, and updating the application state, with

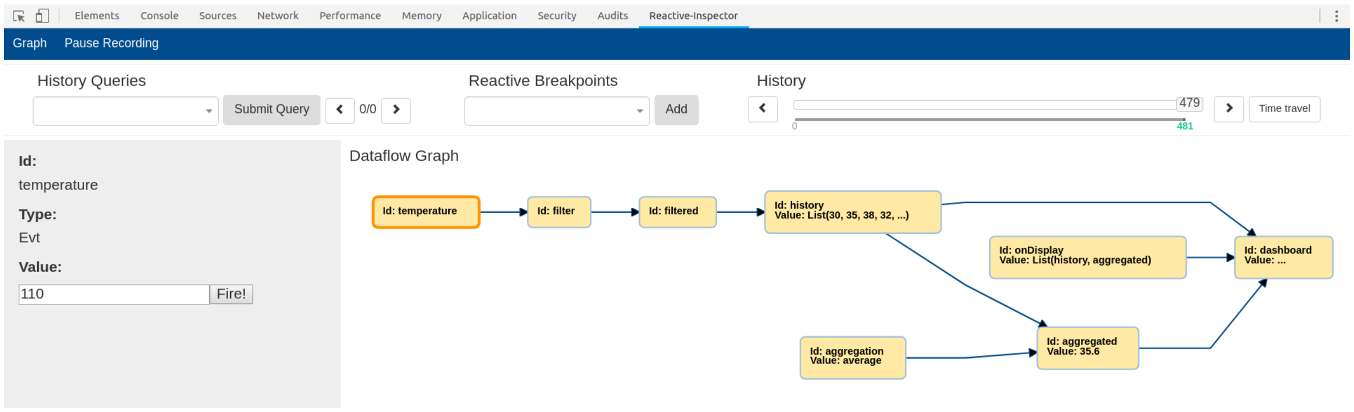

Figure 3: Reactive Inspector with extensions to modify the dataflow graph.

the current tools "what if" questions are surprisingly hard to answer. Triggering an interesting (high temperature) event at the right point in time (history contains 50 elements) requires modifications to the application code to detect the desired event pattern and then trigger the event. The dataflow graph in Reactive Inspector allows one to answer such questions. We extend this tool to enable direct manipulation of the values in the dataflow graph.

Graph Modifications. Figure 3 shows the extended interface of Reactive Inspector. The interface allows changing arbitrary values in the dataflow graph, similar to how interactive debuggers allow changing values assigned to variables. However, changing arbitrary variables of an application may cause inconsistencies in the data model. Reactive Inspector, on the other hand, uses the same mechanism as if the application programmatically changed its input events (`Evt`) or signals (`Var`). Thus, using the dataflow graph prevents inconsistencies by ensuring that each change is fully processed by the application. For example, when the developer issues a new temperature event using the "Fire!" button in the left hand pane of Figure 3, then the new event will be automatically added to the history of events. The developer can then easily answer the question above and conclude that the event did not change the history – it was filtered.

Exposing the graph through the debugger in such a principled manner allows developers to change the code without invalidating basic assumptions of the application. For more flexible debugging purposes, however, we also allow developers to modify arbitrary signals and events, not just inputs (`Var` and `Evt`). For example, the debugger allows one to directly set the average temperature value, even if the new value is no longer reflects the current history of temperature events. Similar to updates of input values, updates to intermediate values are propagated along the dataflow graph, e.g., the dashboard will be updated with the new average. However, forced updates result in visible inconsistencies, such as the dashboard displaying a history with a wrong average. We allow these forced updates as a tool for developers, which are expected to understand the application well enough to prevent accidental inconsistencies.

Time Travel. A well-known technique of debuggers to modify the state of a reactive applications is time traveling [1], i.e., loading a past state of the dataflow graph. REScala has efficient support for snapshotting the dataflow graph [17] and the debugger can reinstate an old state of the application by loading a snapshot. This approach enables users not only to inspect the history of an application, but also to have the actual application jump back and forth in time to enable user interaction with the application in a specific state. Time travel uses the history navigation to select the desired point in time. Time travel is especially useful for "what if" scenarios. The debugger is able to reset the application to a past state – allowing the developer to explore what led to the current state – and modify past values. Past modifications may change the current application state in the future. It is the developer's responsibility not to cause inconsistencies when time traveling.

5 TOWARDS A LIVE TUNING FRAMEWORK

Inspection and modifications of the dataflow graph is not only useful to developers, but also to enable customization – tuning – of the application by domain experts. In this section, we envision a live tuning framework built on top of Reactive Inspector.

Domain experts are assumed to have little or no knowledge of application development, but need to comprehend and modify existing applications. For this use case, we consider correctness and ease of tuning more important than the flexibility of a full-fledged programming language. Domain experts should be able to understand, learn from, and adapt examples, but without risk of breaking the applications. The goal of our framework is to provide limited tuning for most applications out of the box, without additional developer effort. However, for flexible tuning of an applications the framework provides features which developers have to explicitly use. The goal is to enable expert developers to write maintainable live applications (and libraries) at reasonable development effort. Flexibility is achieved through dedicated language abstractions for live tuning for modifying values, rewiring the dataflow graph, and fully customizable developer-defined tuning mechanisms. We describe the spectrum of our abstractions supporting live tuning going from the most simple (ease of use) to the most flexible.

Tunable Values. The first step is to enable domain experts to modify input signals (Var) and trigger input events (Evt). Interacting with inputs is considered safe, as the dataflow graph never has control over external inputs, and has to deal with arbitrary values in any case. Reactive Inspector ensures that users may only enter values of the correct type, and provides easy to use UI abstractions for custom values, e.g., integer inputs are set via sliders, and string inputs with a text box.

However, for some inputs such as the aggregation function of Figure 1 Reactive Inspector is unable to enable easy modifications, because the different aggregation schemes are arbitrary Scala functions. To save domain experts from writing Scala code and to make live modifications accessible, we allow developers to provide predefined behaviors from which the domain experts select. The reactive debugger then displays a drop-down list of possible choices. In the example, the developer could provide an average and a sum function, as an enumeration of aggregation schemes:

```
enum Aggregate(val name: String,
               val compute: Seq[Double] => Double) {
  case Average extends Aggregate("average", ...)
  case Sum extends Aggregate("sum", ...)
}
```

The code snippet implements such choices for the aggregation scheme by using an enumeration providing the the name and the value (in the example, the aggregation function) for every option.

Rewiring Dataflow. A very powerful feature of the dataflow graph is to rewire it at run time. For example, the application developer may want to allow the domain experts to adapt the contents of a dashboard to add or remove displayed elements. The example in Figure 1 already has basic support for such a use case but the domain expert has to modify the list of signals that should be on display. There is a lack of visual collections between the list of signals on display, and the visualized nodes in the dataflow graph. We plan to extend Reactive Inspector to enable direct disconnection and reconnection of edges in the dataflow graph, if a signal involving higher order connections is detected. Connections are only possible between signals with correct types, and the developer can limit connections to a subset of all possible signals.

Developer-defined Tuning. At the end of the spectrum for application tuning are solutions defined by the developer. For example, developers may want to allow the domain experts to create new elements in the dashboard. However, new elements should be limited to a custom domain-specific language designed for data analysis, to make it (a) simple for the domain experts to write and understand such programs, and (b) limit the possible inputs and results to well-defined sets to enforce application properties (i.e., data aggregations should only return numeric results, not fire missiles).

The developer is free to specify how domain experts may interactively tune the application. However, it is then also the developer's responsibility to map the such tuning mechanisms to the underlying dataflow graph modifications (e.g., changed nodes, changed code within nodes, changed edges). The runtime will still take care of ensuring propagation of updates and keeping the application state consistent. We believe that using reactive programming and the dataflow model provides a strong basis for a framework for writing tunable applications, where the concrete run time behavior is interactively developed, e.g., a data analysis application where domain experts can interactively build and combine queries.

6 RELATED WORK

Other debugging approaches more fitting to different high-level approaches have been proposed. In object-centric debugging [11, 18] breakpoints are set on access or change of a specific objects fields, not only on all instances of a certain class, similar to how Reactive Inspector sets breakpoints on specific nodes in the dataflow graph. Lazy evaluation also has the complication of non-sequential execution order like reactive code. Addressing non-sequential evaluation also led to specialized debugging techniques such as oracle debugging [9], where the user interacts with the debugger in a dialog style using questions and answers. Another approach is to record evaluations in a lazy program and then use a strict debugger to inspect the records [8, 14], similar to our use of the chrome debugger for sequential code in the dataflow graph.

A common feature of advanced debugging is to the ability to record the execution and inspect the log afterwards. Clematis [4] is such a tool to record all input events (user actions, incoming server messages, timeouts) and outputs (DOM changes, network request) in a web application. Sahand [3] is an extension to Clematis and additionally records all server inputs (incoming client messages, etc.). These events are correlated and visualized as stories – a graphical representation of the execution not unlike our dataflow graph. However, these stories are sequential in nature and thus not as easy to follow as the dataflow graph, and also do not lend themselves to modifications. Barr et all [6, 7] record execution logs of imperative applications. and use those logs to enable forward and backward navigation in a visualizing debugger similar to our history navigation. By taking advantage of information in the garbage collector about the running program, they reduce the amount of memory needed to record. However, in contrast to the dataflow graph, the state of the imperative application can not be changed after recording, prohibiting modifications. Rxfiddle [5] allows one to visualize the marble diagram and the dataflow of a reactive application. They have not considered modifications.

There are some tools that add live modifcation to applications: The Elm Debugger [1] allows one to go back in the past. They allow modifications, but these have to be done on the source code. Kato and Goto [15] have presented the idea of live tuning where users are presented with a simplified interface – only the tuning sliders – of a full fledged live programming IDE. We use this idea to provide live tuning based on an extended debugger instead of a full IDE, and use reactive programming to make the tuning robust and easy to implement. ZenSheet [2] pushes spreadsheets towards general purpose programming, whereas reactive programming can be described as spreadsheet semantics for general purpose programming languages. Our approach has the advantage that existing tools, libraries, and infrastructure can be reused, however at the cost of less flexibility at runtime. We believe that both approaches provide their own value, and it will be interesting to see how far they can push the boundaries in the corresponding directions.

7 CONCLUSION

We have shown that reactive programming facilitates writing of tunable live applications by using the existing dataflow graph structure of reactive programs. The dataflow graph is a suitable abstraction for both writing reactive source code and representing reactive code visually to interactively inspect, change and evolve the behavior of an application. The presented approach to modify values in the graph is an initial step towards making the graph more tunable through live modifications. Augmenting reactive programming with abstractions specific for tuning allows for safe run time modifications also by non-expert developers. In perspective, we want to support and to be able to mix both forms of "traditional" development (compiling and re-executing) and "live" development (changing code during run time), so developers can freely choose to what degree they use which approach to solve specific problems during the development process.

ACKNOWLEDGMENTS

This work is supported by the European Research Council (ERC, Advanced Grant No. 321217), by the German Research Foundation (DFG, SFB 1053 and SA 2918/2-1), and by the LOEWE initiative in Hessen, Germany (HMWK, NICER).

REFERENCES

[1] 2018. Elm's Time Traveling Debugger. http://debug.elm-lang.org/.
[2] Enzo Alda and Monica Figuera. 2017. ZenSheet: a live programming environment for reactive computing. In *Workshop on Live Programming Systems (LIVE)*.
[3] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding asynchronous interactions in full-stack JavaScript. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 1169–1180. https://doi.org/10.1145/2884781.2884864
[4] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding JavaScript Event-Based Interactions with Clematis. *ACM Trans. Softw. Eng. Methodol.* 25, 2 (2016), 12:1–12:38. https://doi.org/10.1145/2876441
[5] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data Flows in Reactive Programs. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. https://doi.org/10.1145/3180155.3180156
[6] Earl T. Barr and Mark Marron. 2014. Tardis: affordable time-travel debugging in managed runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 67–82. https://doi.org/10.1145/2660193.2660209
[7] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. 2016. Time-travel debugging for JavaScript/Node.js. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 1003–1007. https://doi.org/10.1145/2950290.2983933
[8] Bernd Braßel. 2009. A Technique to Build Debugging Tools for Lazy Functional Logic Languages. *Electr. Notes Theor. Comput. Sci.* 246 (2009), 39–53. https://doi.org/10.1016/j.entcs.2009.07.014
[9] Bernd Braßel and Holger Siegel. 2007. Debugging Lazy Functional Programs by Asking the Oracle. In *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*. 183–200. https://doi.org/10.1007/978-3-540-85373-2_11
[10] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-value Language. In *Proceedings of the 15th European Conference on Programming Languages and Systems (ESOP)*. https://doi.org/10.1007/11693024_20
[11] Claudio Corrodi. 2016. Towards Efficient Object-Centric Debugging with Declarative Breakpoints. In *Post-proceedings of the 9th Seminar on Advanced Techniques and Tools for Software Evolution, Bergen, Norway, July 11-13, 2016*. 32–39. http://ceur-ws.org/Vol-1791/paper-04.pdf
[12] Conal Elliott. 2009. Push-pull functional reactive programming. *Proc. 2nd ACM SIGPLAN Symp. Haskell - Haskell '09* (2009). https://doi.org/10.1145/1596638.1596643
[13] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/258948.258973
[14] Robert Ennals and Simon L. Peyton Jones. 2003. HsDebug: debugging lazy programs by not being lazy. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003*. 84–87. https://doi.org/10.1145/871895.871904
[15] Jun Kato and Masataka Goto. 2016. Live Tuning: Expanding Live Programming Benefits to Non-Programmers. In *Workshop on Live Programming Systems (LIVE)*.
[16] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. https://doi.org/10.1145/1640089.1640091
[17] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:26. https://doi.org/10.4230/LIPIcs.ECOOP.2018.1
[18] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-centric debugging. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 485–495. https://doi.org/10.1109/ICSE.2012.6227167
[19] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY)*. https://doi.org/10.1145/2577080.2577083
[20] Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. https://doi.org/10.1145/2884781.2884815
[21] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini. 2017. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering* 43, 12 (Dec 2017). https://doi.org/10.1109/TSE.2017.2655524