Live Programming for Event-Based Languages

Short Paper

Christopher Schuster
University of California
Santa Cruz
cschuste@ucsc.edu

Cormac Flanagan
University of California
Santa Cruz
cormac@ucsc.edu

ABSTRACT

Live programming environments assist programmers by allowing code edits to running programs, providing continuous feedback and potentially even traveling back in time to past execution states. Event-based languages like JavaScript facilitate these features, but the entanglement of code, state and output still hinders live programming. This paper shows how hot swapping, time travel and continuous feedback can be achieved by restricting standard JavaScript programs to have a single, pure rendering function and no function values in the global state. Furthermore, we describe this design for general event-based languages and how these properties can be enforced statically or dynamically.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

JavaScript, Live Programming, Event Handling, Debugging

1. INTRODUCTION

Programming can be difficult due to the mismatch between the programmer's intuition and the actual execution. To make programming easier, the language and environment should reduce the mental and time gap between code edits and visible feedback which is a core idea of *live programming.*

Live edits to running programs (*hot swapping*) have been explored before, e.g. in the context of Dynamic Software Updating (DSU)[10]. Event-based languages simplify these updates as there is no active call stack before and after events.

1.1 Challenges

An unavoidable challenge of live programming is the inherent entanglement between state and code. Code updates in the presence of these dependencies may require the programmer to specify data conversions which delays feedback and thereby hinders live programming.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
REBLS'15 October 23, 2015, Pittsburgh, USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

Another obstacle for live programming are function values/closures which are stored in the application state. These need to be updated in order to avoid stale code. However, nested closures with scoped variable references and non-trivial mappings between old and new function literals complicate this process.

Finally, the visible feedback could become outdated if code for generating the output (e.g. the user interface) is not re-executed after a hot swapping operation. If the output is part of the application state, code updates would therefore also involve non-trivial conversions to refresh the output.

1.2 Usable Live Programming

Burckhardt et. al. [4] and McDirmid [14] presented a solution to these challenges for reactive event based systems. In contrast to DSU systems, which guarantee safe updates with the cost of manual data conversions, a *best effort* approach better suits the goals of live programming.

To prevent outdated code in closures and avoid complicated transformations of these closures, the global application state is restricted to not contain any function values.Furthermore, if the rendering code is separated from event handling and does not mutate state, it can be evaluated continuously to keep the displayed output up-to-date.

This short paper uses the same approach to achieve hot swapping and time travel but, in contrast to earlier work,

- we describe a solution for plain JavaScript without special language constructs and independent of the concrete rendering approach/framework, and
- we generalize this approach to any event-based language and briefly discuss ways to ensure the aforementioned properties statically and dynamically.

2. LIVE PROGRAMMING IN JAVASCRIPT

To illustrate the approach, it is useful to first consider a 'traditional' JavaScript application.

2.1 Traditional JavaScript Event Handling

The example in Figure 1 uses the jQuery function '`$`' to select DOM elements, attach an event handler for clicks on button `b` and manipulate the DOM. Independent of coding style, this imperative way of changing the user interface and registering event handlers hinders live programming.

Live edits to the code, e.g. renaming "Count:" to "Clicks:", would require updating the closure stored in the DOM state and even then the edit would not result in an updated visible output until the next time the event handler executes.

```
<div id="c">Count: 0</div>
<button id="b">Inc!</button>
<script>
  var i = 0;
  $("#b").on("click", function() {
    $("#c").html("Count: " + (++i));
  });
</script>
```

Figure 1: JavaScript Example with DOM Manipulation and Callback Event Handling.

2.2 Separating Rendering and Event Handling

In order to solve the problems outlined above, we propose to separate the program into three separate functions:

- `init()` returns the initial application state.
- `handle(event,state)` processes an event based on the previous state and returns the new application state.
- `render(state)` returns the output/DOM based on the current state without modifying the global state.

Additionally, the state returned by `init()` and `handle()` cannot contain closures (as discussed above).

This style allows any state to be visualized with `render()` including previous states to enable a simple form of time travel ("debugging back in time" [12]). Furthermore, any function can be hot swapped with an updated version and both the output and the subsequent event handling will adhere to the updated code without influence from outdated closures or event handlers.

```
function init() { return {i: 0}; }
function handle(evt, state) {
  if (evt instanceof MouseEvent &&
      evt.target.id === "b") {
    return {i: state.i + 1};
  } else {
    return state;
  }
}
function render(state) {
  return "<div>Count: " + state.i + "</div>" +
         "<button id='b'>Inc!</button>";
}
```

Figure 2: Separating rendering and event handling.

2.3 Implicit style

The approach outlined above does not fit the common JavaScript coding style of callbacks as event handlers and updating state with simple assignments.

Fortunately, manual event dispatching can be avoided by defining the set of active event handlers for each element declaratively while avoiding imperative DOM manipulation.

Additionally, global variables can be used in lieu of an explicit state value. This enables standard variable initialization instead of an explicit `init` function and standard state mutation instead of returning the new state.

To support function values in the output, `render()` now returns a tree structure instead of a plain string. The details of how this tree structure is defined and constructed is application-specific and insignificant for live programming. In Figure 3, we use inline XML/HTML tags to create a tree representation of the output (also known as JSX syntax).

```
var i = 0;
function inc() { ++i; }
function render() {
  return (
    <div>Count: {i}
      <button onclick={inc}>Inc!</button>
    </div>);
}
```

Figure 3: Implicit style for the code in Figure 2.

$$\begin{array}{llr}
e ::= \lambda x.\ e \ \mid\ e(e) \ \mid\ x \ \mid\ \{x : e, ...\} \ \mid\ ... & & \text{(Expressions)} \\
v ::= \lambda x.\ e \ \mid\ \{x : v, ...\} \ \mid\ ... & & \text{(Values)} \\
p ::= \{\text{init} : e,\ \text{handle} : e,\ \text{render} : e\} & & \text{(Programs)} \\
q ::= [\text{event } v] \mid [\text{swap } p] \mid [\text{reset}] \mid [\text{time } n] & & \text{(Events)} \\
e \downarrow v \quad \text{(Evaluation)} & \langle q, p, \vec{S}\rangle \Downarrow \langle p', \vec{S}', O\rangle & \text{(Updates)}
\end{array}$$

$$\frac{p.\text{handle}(v, S_j) \downarrow S_{j+1} \qquad p.\text{render}(S_{j+1}) \downarrow O}{\langle [\text{event } v], p, (..., S_j)\rangle \ \Downarrow\ \langle p, (..., S_j, S_{j+1}), O\rangle}\ \text{E-EVENT}$$

$$\frac{p'.\text{render}(S_j) \downarrow O}{\langle [\text{swap } p'], p, (..., S_j)\rangle \ \Downarrow\ \langle p', (..., S_j), O\rangle}\ \text{E-SWAP}$$

$$\frac{p.\text{init} \downarrow S_0' \qquad p.\text{render}(S_0') \downarrow O}{\langle [\text{reset}], p, \vec{S}\rangle \ \Downarrow\ \langle p, (S_0'), O\rangle}\ \text{E-RESET}$$

$$\frac{p.\text{render}(S_n) \downarrow O}{\langle [\text{time } n], p, (..., S_n, ...)\rangle \ \Downarrow\ \langle p, (..., S_n), O\rangle}\ \text{E-TIME}$$

Figure 4: Operational semantics for event handling, hot swapping and traveling back in time.

3. FORMALISM

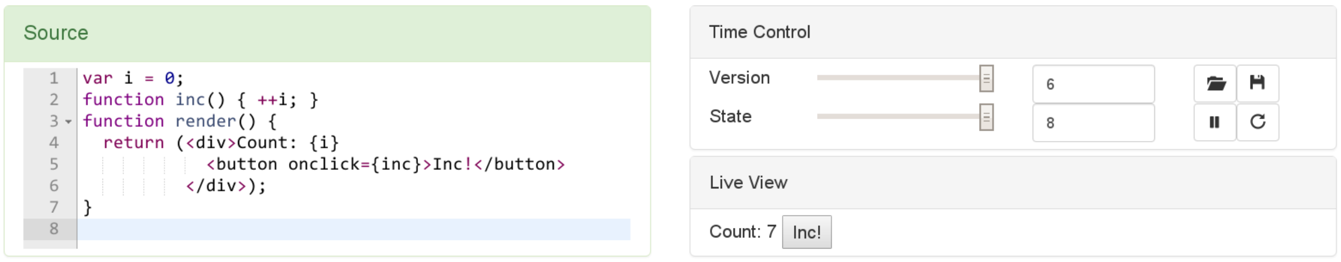
The live programming system described in the previous section can be generalized to any event-based language whose programs follow a certain top level structure.

Figure 4 describes the semantics for event handling, hot swapping and time traveling, while leaving most of the underlying language unspecified. Assuming that functions are expressions e in the language, a valid program p simply consists of three functions named 'init', 'handle' and 'render'. A system configuration then consists of a program p and a sequence of application states $\vec{S}$, which are simple values v in the base language. Application-specific events [event v] are passed to the current 'handle' function alongside the most recent state S_j, yielding a new state S_{j+1}; [swap p'] replaces the current program p; [reset] restarts execution using the current 'init' function; and [time n] reverses the last n execution steps. All state transitions update the output O based on the current 'render' function and most recent state.

To guarantee that the output is up-to-date with the current state and that no stale code persists after a code update,

- the 'render' function cannot change the state, and
- the state after 'init'/'handle' cannot contain functions.

These two properties can be checked dynamically by a contract that ensures that the state before 'render' and after 'render' is identical, and a *first-order contract* that checks the state after 'init' and 'handle' for functions.



```
var i = 0;
function inc() { ++i; }
function render() {
  return (<div>Count: {i}
          <button onclick={inc}>Inc!</button>
        </div>);
}
```

Figure 5: The live programming environment features an editor, a live view of the output as well as controls for traveling to previous code versions/execution states and for reseting the state to initial values.

Alternatively, these properties can also be checked statically by a type and effect system. This avoids the performance overhead of dynamic checking but might reject valid programs. While mostly preferable for languages with type annotations, sophisticated analysis methods can also check unannotated, dynamically-typed languages [16, 21].

4. IMPLEMENTATION

The programming environment is shown in Figure 5. Its source code[1] and a live demo[2] are both publicly available.

Each change in the editor causes the code to get parsed, checked, rewritten and evaluated. Global variable references `x` are rewritten to `state["x"]` and global variable initializers extracted to a separate `init()` function. This is necessary in order to guard the state against mutation during rendering with a recursive `Object.freeze`, to deep clone the state at each event for time travel and to check the state for closures. Additionally, all event handlers in the output of `render()` are wrapped to re-render and refresh the live view.

5. RELATED WORK

5.1 Reactive Programming

There has been an increased interest in applying functional reactive programming (FRP) to web programming recently. While these projects are primarily focussing on user interfaces (e.g. Elm [6], KScript/KSWorld [17] and React) or FRP primitives (e.g. Bacon.js, RxJS), they are perfect candidates for live programming environments.

More generally, a case study by Kambona et. al. compares reactive programming with Promises and plain event handling in JavaScript in JavaScript [11]. To bridge the client/server separation in web programming, both Reynders et. al. and Chlipala present unified/multi-tier systems with FRP/data flow aspects [5, 18]. Instead of introducing a unified language or language features for FRP, this paper focuses on standard, single-threaded JavaScript applications.

Combining FRP with imperative/OO programming opens a space of design decisions with different trade-offs [20]. In contrast to REScala [3, 19], which integrates FRP signals and event handling, and the recent work of Zhuang and Chiba [23], this paper enforces a pure FRP style at the top level while supporting regular OO programming and declarative binding of event handlers with the "implicit style".

This project would benefit from *self-adjusting computation* [1, 2], which reactively recomputes incremental changes to the output based on changes in the inputs. Currently, the output gets completely recomputed at change in state.

[1]Source code at http://github.com/levjj/rde/
[2]Online live demo at http://levjj.github.io/rde/

5.2 Dynamic Software Updating (DSU)

Hot swapping a new code is closely related to research on dynamic software updating (for a survey, see Seifzadeh et. al. [22]). As shown by Fabry, hot swapping a module is straight-forward if there is no persistent state but otherwise requires data transformations [8]. Later Hicks described a safe, typed DSU system which enables the programmer to provide patches for these transformations [10]. Hot swapping as described in this paper tries to avoid manual intervention in favor for a best effort approach since application state can also be interactively reset during development.

5.3 Live Programming

Live programming with continuously updated feedback was first established by Hancock [9] and further explored by systems like SuperGlue, which uses dynamic inheritance and explicit FRP signals [13], and Elm, which demonstrates live programming and time traveling with first-order FRP. The work by both Burckhardt et. al. and McDirmid is probably most closely related to this project as it showed how a pure render function and an application state without closures facilitates live programming [4] and outlined the possible design space between live programming systems that resume computation with a possibly inconsistent state and a systems that record replay execution [14].

Finally, Glitch is an example of using imperative updates that propagate until coherence (similarily to the Coherence language by Edwards [7]) in a system that supports both live programming and time travel [15].

6. CONCLUSION AND DISCUSSION

This paper showed that JavaScript supports hot swapping, time traveling and continuous feedback by enforcing a pure rendering function on the top level and excluding closures from the application state. This concept can be generalized to any event-based language and does not depend on a specific user interface framework.

However, restricting the global state in this way limits expressiveness and excludes classic OOP objects. More research is necessary to evaluate the design decisions and study how programmers can benefit from live programming.

Future work might also involve programming language techniques like proxy membranes for enforcing immutability and first-order states, for optimizing time travel with copy-on-write, and for incrementally computing the new output instead of recomputing the complete output after each event.

7. REFERENCES

[1] U. A. Acar. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pages 1–6, New York, NY, USA, 2009. ACM.

[2] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 247–259, New York, NY, USA, 2002. ACM.

[3] E. G. Boix, K. Pinte, S. Van de Water, and W. D. Meuter. Object-oriented reactive programming is not reactive object-oriented programming. *REM'13*, 2013.

[4] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! continuous feedback in ui programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 95–104, New York, NY, USA, 2013. ACM.

[5] A. Chlipala. Ur/web: A simple model for programming the web. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 153–165, New York, NY, USA, 2015. ACM.

[6] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM.

[7] J. Edwards. Coherent reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 925–932, New York, NY, USA, 2009. ACM.

[8] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 470–476, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[9] C. M. Hancock. *Real-time Programming and the Big Ideas of Computational Literacy.* PhD thesis, Cambridge, MA, USA, 2003. AAI0805688.

[10] M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 13–23, New York, NY, USA, 2001. ACM.

[11] K. Kambona, E. G. Boix, and W. D. Meuter. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, New York, NY, USA, 2013. ACM, ACM.

[12] B. Lewis and M. Ducasse. Using events to debug Java programs backwards in time. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 96–97, New York, NY, USA, 2003. ACM.

[13] S. McDirmid. Living it up with a live programming language. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 623–638, New York, NY, USA, 2007. ACM.

[14] S. McDirmid. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 53–62, New York, NY, USA, 2013. ACM.

[15] S. McDirmid and J. Edwards. Programming with managed time. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 1–10, New York, NY, USA, 2014. ACM.

[16] J. Nicolay, C. Noguera, C. D. Roover, and W. D. Meuter. Detecting function purity in javascript. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, SCAM '15, Sept 2015.

[17] Y. Ohshima, A. Lunzer, B. Freudenberg, and T. Kaehler. Kscript and ksworld: A time-aware and mostly declarative language and interactive gui framework. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 117–134, New York, NY, USA, 2013. ACM.

[18] B. Reynders, D. Devriese, and F. Piessens. Multi-tier functional reactive programming for the web. In *Onward! 2014*, pages 55–68. ACM, October 2014.

[19] G. Salvaneschi, G. Hintz, and M. 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, New York, NY, USA, 2014. ACM.

[20] G. Salvaneschi and M. Mezini. Reactive behavior in object-oriented applications: An analysis and a research roadmap. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 37–48, New York, NY, USA, 2013. ACM.

[21] C. Schuster and C. Flanagan. A light-weight effect system for javascript. In *Proceedings of the 2015 Scripts to Programs Workshop*, STOP '15, July 2015.

[22] H. Seifzadeh, H. Abolhassani, and M. S. Moshkenani. A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568, 2013.

[23] Y. Zhuang and S. Chiba. Enabling the automation of handler bindings in event-driven programming. pages 350–360, July 2015.