# Reactive Programming with Reactive Variables

Christopher Schuster Cormac Flanagan

University of California, Santa Cruz
Santa Cruz, CA 95064, USA
{cschuste,cormac}@ucsc.edu

## Abstract

Reactive Programming enables declarative definitions of time-varying values (signals) and their dependencies in a way that changes are automatically propagated. In order to use reactive programming in an imperative object-oriented language, signals are usually modelled as objects. However, computations on primitive values then have to lifted to signals which usually involves a verbose notation. Moreover, it is important to avoid cycles in the dependency graph and glitches, both of which can result from changes to mutable global state during change propagation.

This paper introduces *reactive variables* as extension to imperative languages. Changes to reactive variables are automatically propagated to other reactive variables but, in contrast to signals, reactive variables cannot be reified and used as values. Instead, references to reactive variables always denote their latest values. This enables computation without explicit lifting and limits the dependents of a reactive variable to the lexical scope of its declaration. The dependency graph is therefore topologically ordered and acyclic. Additionally, reactive updates are prevented from mutating global state to ensure consistency. We present a working prototype implementation in JavaScript based on the sweet.js macro system and a formalism for integration with general imperative languages.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***Keywords*** Reactive Programming, Syntax Extension, JavaScript

## 1. Introduction

Reactive Programming [1] is a programming paradigm which has recently attracted more attention due to its benefits for programming user interfaces. In contrast to imperative programming languages, reactive languages do not evaluate a program statement by statement. Instead, values are recomputed whenever their inputs are updated. Thereby, the program can be modelled as *signal-flow graph* in which the nodes are *signals* or *behaviors*, i.e. time-varying values, and change is propagated along the edges. The declarative specification of dependencies and update expressions essentially corresponds to *unidirectional constraints*.

One challenge of reactive programming is the integration with imperative code. While libraries can provide data structures for signals with reactive change propagation, these library often require user code to be *lifted* onto signals which involves significant syntactic overhead. Furthermore, the integration with imperative code can also lead to dependency cycles and *glitches* [2, 3].

```
 1 var count = 0;
 2 var paused = false;
 3 setInterval(function() {
 4   if (!paused) {
 5     count++;
 6     $("#countBtn").text("Count:" + count);
 7   }
 8 }, 100);
 9
10 $("#countBtn").click(function() {
11   paused = !paused;
12   $("#countBtn").text(
13     paused ? "Paused" : "Count: " + count);
14 });
```

**Figure 1.** JavaScript code with callbacks and imperative updates.

This paper proposes *reactive variables* as alternative approach to reactive programming. Overall, the contributions of this short paper are

- a new language feature called *reactive variables*,
- an implementation for JavaScript based on the sweet.js macro system [5], and
- formal operational semantics for adding reactive variables to an imperative language.

## 2. Existing Approaches

To motivate the need for reactive programming, let us consider a common use case of an interactive user interface whose output depends on multiple inputs.

### 2.1 Imperative Updates

The JavaScript application shown in Figure 1 implements a counter which can be paused and resumed, and is displayed as text. There are two sources of change:

1. a timing event which is triggered every 100 milliseconds, and
2. button clicks by the user.

The main disadvantage of this imperative approach is that both events are handled with callbacks that modify shared state and update the user interface which results in duplicated code in lines 6 and 12-13.

```
 1 var pausedSignal = Rx.Observable
 2   .fromEvent($("#countBtn"), "click")
 3   .scan(function(cnt) {return !cnt;}, false)
 4   .startWith(false);
 5
 6 Rx.Observable.interval(100)
 7   .pausable(pausedSignal)
 8   .scan(function(c) { return c + 1; }, 0)
 9   .startWith(0)
10   .combineLatest(pausedSignal,
11     function(count, paused) {
12       return paused ? "Paused"
13                     : "Count: " + count; })
14   .subscribe(function(s) {
15     $("#countBtn").text(s); });
```

**Figure 2.** Implementation using the RxJS library (Reactive Extensions for JavaScript).

### 2.2 Libraries for Reactive Programming

A common approach to enable reactive programming in object-oriented languages is to use a library with data structures for creating and combining signals.

Figure 2 illustrates how the code in Figure 1 could be rewritten with the RxJS library[1] which provides reactive extensions for JavaScript. Here, `pausedSignal` is a signal of boolean values that alternate with each button click and has an initial value of `false`. The second signal is based on sampling every 100 milliseconds and can be paused depending on the current value of the first signal. If not paused, it increments a counter starting with 0. Finally, the `combineLatest` operator causes any change in either of the two signals to update the button label.

The main disadvantage of this solution is the verbose syntax for expressing the signal-flow graph. Primitive operations cannot operate on signals directly, so they have to written as functions and passed into the library, e.g. in lines 3, 8, 12-13 and 15. Moreover, functions passed into the update can still reference and update global state. This can lead to non-reactive dependencies on state that are hard to reason about, as illustrated by the following code:

```
var x = 1;
Rx.Observable.interval(100)
  .map(function() { return x * x; })
  .map(function(x2) { x++; return x2; }); // !
  .subscribe(function(x2) { alert([x, x2]); });
// Output is  [2,1], [3,4], [4,9], [5,16] ...
// instead of [1,1], [2,4], [3,9], [4,16] ...
```

## 3. Reactive Variable Declarations

The need to lift primitive operations when using reactive programming libraries stems from the use of special signal values to model streams of values/events [2].

In this paper, we propose to model dependencies with *reactive variables* that cannot be reified, so all values in the language are primitive and do not require lifting. In contrast to reactive values (signals), a reactive variable can only be referenced in a limited, lexical scope which is declared with `rlet`.

A reactive variable reference always evaluates to its current/last value. Additionally, a reference used in another reactive variable declaration, or explicitly as part of a subscription, denotes a dependency such that updates are automatically propagated.

[1] `https://github.com/Reactive-Extensions/RxJS`

[2] A good type system might help to avoid confusing primitive values with signal (see e.g. Elm[4]) but still requires lifting.

Only reactive variables in the lexical scope can be referenced, therefore there can be no cyclic dependencies. Furthermore, reactive variables sorted by their lexical scope are also topologically ordered regarding their dependencies, so propagating changes from outer to inner reactive variable definitions cannot cause a variable to change more than once (which would be considered a *glitch* [2]).

### 3.1 Example

The example code in Figure 3 illustrates how reactive variables support reactive programming. The example implements the same application as in Figure 1 and 2 and its dependencies graph resulting from nested reactive variables declarations is shown in Figure 4.

```
1 rlet paused = subscribe($("#countBtn").click)
2               initially(false) !paused;
3 rlet count = subscribe(interval(100))
4              initially(0)
5              paused ? count : count + 1;
6 rlet txt =
7   paused ? "Paused" : "Count: " + count;
8 subscribe(txt) { $("#countBtn").text(txt); }
```

**Figure 3.** First-class reactive variables with `rlet`.

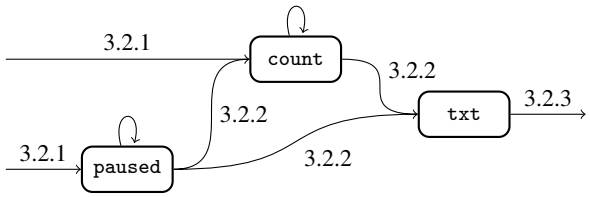


**Figure 4.** Dependency graph for reactive variables shown in Figure 3. The edges are labeled with the feature used for change propagation as described in Section 3.2.

### 3.2 Evaluation and Change Propagation

Reactive variables and their dependencies can be declared with `rlet` but the actual change propagation will always be initiated by imperative updates, which cause a recomputation of all outdated reactive variables, before the imperative evaluation continues. This simple evaluation technique is therefore *synchronous* but more sophisticated systems could also support asynchronous change propagation in which multiple updates can overlap.

#### 3.2.1 Triggering a Reactive Update

If `txt` is a reactive variable in scope, the imperative `push` syntax can be used to assign a new value and automatically propagate the change, ignoring its update expression[3], e.g.

```
rlet txt;
push(txt, "foo");  // or just 'txt = "foo";'
```

In addition to simple 'push updates', reactive variable declarations also provide syntactic sugar to subscribe to standard JavaScript event sources that expect a callback function:

```
rlet txt = subscribe(input.changed) input.text;
```

[3] This simplifies imperative updates but violates the constraint character of reactive variables. Alternatively, the system could differentiate between reactive variables and imperative event sources (which have no update expressions).

### 3.2.2 Propagating Changes to Dependent Variables

As explained above, referencing other reactive variables in a declaration automatically sets up a dependency. It is also possible to manually subscribe to another reactive variable in order to react to its changes independent of its value.

```
rlet len = txt.length; //implies subscribe(txt)
rlet lastChanged = subscribe(txt) Date.now();
```

Global variables cannot be mutated during change propagation. However, it is still possible to support stateful computation by referencing a reactive variable in its own update expression which then denotes the previous value (an explicit initial value `i` can be supplied for the first update)[4]:

```
rlet numChanged = subscribe(txt)
                  initially(0) numChanged + 1;
```

### 3.2.3 Reacting to Updated Variables

After all changes have been propagated according to the dependency graph, the normal imperative evaluation resumes and reactive variables might have new values.

```
console.log(numChanged);    // "23"
push(txt, "bar");
console.log(numChanged);    // "24"
```

Additional syntax could be added to execute imperative code in response to reactive changes:

```
subscribe(numChanged) { console.log("goo"); }
push(txt, "bar");           // "goo"
```

## 4. Implementation

Instead of implementing a custom language runtime, we added reactive variables to JavaScript as a syntax extension based on the sweet.js macro system [5]. Macros help to provide a better syntax for reactive programming compared to a pure library approach (see Section 2.2) while remaining compatible to the remaining JavaScript ecosystem.

Figure 5 illustrates the macro expansion of reactive variable declarations to regular JavaScript code. Here, the `Signal` class is used for modelling signals, similarly to RxJS and Flapjax [10]. The first argument of the constructor is a list of signals to subscribe to, the second is the update expression and the third an initial value. In addition to expressions and reactive variables explicitly listed in `subscribe`, the macro expansion will also include all reactive variables referenced in the update expression as dependency.

In order to avoid state mutation by the update, all references to variables from the surrounding scopes in the update expression will be replaced by a call to `IMM` (as shown in Figure 5) which wraps the object with an immutability proxy membrane [13, 14] that prevents field updates to the wrapped objects and anything reachable from the wrapped object[5].

Both the source code of the implementation and a live online demo are publicly available [6].

## 5. Formalism

For clarity, we also illustrate the semantics of reactive variables via the operational semantics in Figure 6. Here, the target language is

```
1 // helper signal for subscribe($("#coun...)
2 var _s0 = new Signal([]);
3
4 $("#countBtn").click(function() {
5   push(_s0, null); // click invokes update
6 });
7
8 // signal for "paused"
9 var s_paused = new Signal(
10   [_s0],              // dependencies
11   function() {        // update expression
12     return !IMM(s_paused);
13   },
14   false);             // initial value
```

**Figure 5.** Macro expanded JavaScript code for lines 1-3 of Figure 3 based on a simple 'Signal' class for managing dependencies.

the imperative lambda calculus with reference cells extended with reactive variables.

In addition to the mutable heap $H$, the evaluation environment consists of a signal-flow graph $S$, represented as list of reactive updates $(r(r_s...) \leftarrow e;)...$, and the reactive variables $R$, which are mapped to their respective heap locations $l$.

A reactive variable declaration is evaluated by checking that its dependencies $r_s$ are currently in scope $R$, allocating a new signal with a heap location $l$, and then adding the update expression $r(r_s...) \leftarrow e_2$ to the signal-flow graph $S$.

Evaluating a 'push update' of a reactive variable $r$ assigns a new value to $r$ and then initiates the reactive change propagation which evaluates the current signal-flow graph $S$ with $r$ marked as changed in the set of changed variables $C$.

The reactive change propagation itself evaluates the list of reactive updates $S$ normally but without performing assignments to non-reactive variables. Also, reactive variable updates $r(r_s...) \leftarrow v$ will be ignored if none of the dependencies $r_s...$ have been changed. Otherwise, the reactive variable will be updated with the new computed value $v$ and marked as changed for subsequent updates.

## 6. Related Work

The Functional Reactive Programming (FRP) model was first introduced by Elliot and Hudak [7] as the Fran library for animations in Haskell. It established the term *behavior* for time-varying values with continuous semantics which can be sampled to obtain discrete events. A more modern implementation of FRP called Elm [4] uses *signals* to model both continuous and discrete time-varying values. Signals in Elm are first-order, so the result signal-flow graph itself is static which is also true for reactive variables. While evaluation for reactive variables is assumed to be synchronous, Elm allows multiple changes to propagate concurrently and even enables the evaluation to violate the global order of events, so long-running background computations do not block other signals.

There has been prior work on integrating reactive programming with an imperative, object-oriented programming (OOP). Most notably, Flapjax [10] enabled reactive programming in JavaScript and inspired popular libraries like RxJS (see Section 2.2). Similar to the Fran library, Flapjax differentiates between continuous behaviors and discrete *event streams* and offers many different operators to combine and convert these. These operators are often specifically tailored to facilitate programming of web applications. In contrast to Flapjax, reactive variables have no notion of 'events' and instead interact with the imperative program with a push/subscribe model. Additionally, the Flapjax library requires explicit lifting of primitive operations which can only be avoided by using the Flapjax compiler

[4] Alternatively, there could be special syntax for referring to the previous and current value of a variable in order to retain the constraint character of the reactive variable declaration.

[5] As a side effect, this rewriting turns assignments to global variables into invalid JavaScript, thereby preventing mutation of global state.

[6] Source code and demo at `https://github.com/levjj/rlet`

$$v \quad := () \mid \lambda x.e \mid l \qquad (v : \text{Value},\ x : \text{Regular variable},\ r : \text{Reactive variable},\ e : \text{Expression},\ l : \text{Heap location})$$

$$e \quad := v \mid x \mid r \mid e(e) \mid \text{ref } e \mid e := e \mid !e \mid \text{rlet } r = \text{subscribe}(r...) \text{ initially}(e)\ e \text{ in } e \mid \text{push}(r, e) \mid r(r...) \leftarrow e$$

$$E[\circ] = \circ \mid E(e) \mid v(E) \mid E; e \mid \text{ref } E \mid E := e \mid l := E \mid !E \mid \text{push}(r, E) \mid r(r_s...) \leftarrow E \qquad \text{(Evaluation context)}$$

$$S : (r(r_s...) \leftarrow e;)... \text{ (Reactive Updates)} \quad R : r \to l \text{ (Reactive Variables)} \quad C : \mathcal{P}(r) \text{ (Changed)} \quad H : l \to v \text{ (Heap)}$$

**Pure computation** ($R, H \vdash e \to e$)

$$\frac{}{R, H \vdash (\lambda x.e)(v) \to e[x/v]}\ \text{E-APP} \qquad \frac{}{R, H \vdash (v; e) \to e}\ \text{E-SEQ} \qquad \frac{H(l) = v}{R, H \vdash\ !l \to v}\ \text{E-DEREF} \qquad \frac{R(r) = l \quad H(l) = v}{R, H \vdash r \to v}\ \text{E-RVAR}$$

**Imperative evaluation** ($S, R \vdash H, e \twoheadrightarrow H, e$)

$$\frac{S, R \vdash H, e \twoheadrightarrow H', e'}{S, R \vdash H, E[e] \twoheadrightarrow H', E[e']}\ \text{I-CTX} \qquad \frac{R, H \vdash e \to e'}{S, R \vdash H, e \twoheadrightarrow H, e'}\ \text{I-PURE} \qquad \frac{l \text{ fresh} \quad H' = H[l := v]}{S, R \vdash H, \text{ref } v \twoheadrightarrow H', l}\ \text{I-REF} \qquad \frac{H' = H[l := v]}{S, R \vdash H, l := v \twoheadrightarrow H', v}\ \text{I-ASG}$$

$$\frac{\{r_s...\} \subseteq dom(R) \qquad l \text{ fresh} \qquad S' = S; r(r_s...) \leftarrow e_2 \qquad R' = R[r := l] \qquad S', R' \vdash H, l := e_1; e_3 \twoheadrightarrow^* H', v}{S, R \vdash H, \text{rlet } r = \text{subscribe}(r_s...) \text{ initially}(e_1)\ e_2 \text{ in } e_3 \twoheadrightarrow H', v}\ \text{I-RLET}$$

$$\frac{R(r) = l \qquad C = \{r\} \qquad H' = H[l := v] \qquad R \vdash C, H', S \rightsquigarrow^* C', H'', ()}{S, R \vdash H, \text{push}(r, v) \twoheadrightarrow H'', ()}\ \text{I-PUSH}$$

**Reactive change propagation** ($R \vdash C, H, e \rightsquigarrow C, H, e$)

$$\frac{R \vdash C, H, e \rightsquigarrow C', H', e'}{R \vdash C, H, E[e] \rightsquigarrow C', H', E[e']}\ \text{R-CTX} \qquad \frac{R, H \vdash e \to e'}{R \vdash C, H, e \rightsquigarrow C, H, e'}\ \text{R-PURE} \qquad \frac{C \cap \{r_s...\} = \emptyset}{R \vdash C, H, r(r_s...) \leftarrow v \rightsquigarrow C, H, ()}\ \text{R-UNCHANGED}$$

$$\frac{C \cap \{r_s...\} \neq \emptyset \qquad R(r) = l \qquad C' = C \cup \{r\} \qquad H' = H[l := v]}{R \vdash C, H, r(r_s...) \leftarrow v \rightsquigarrow C', H', ()}\ \text{R-CHANGED}$$

**Figure 6.** Operational semantics for reactive variables in a lambda calculus with reference cells.

which automatically rewrites JavaScript to lift all operations in the program to behaviors.

Other efforts to combine OOP with reactive programming introduce first-class events and language constructs to dispatch and subscribe to events. OOP with first class events is particularly useful for GUI programming as user interfaces are often modelled as objects reacting to events. REScala [12] integrates declarative, reactive values and events handling in Scala but, in contrast to reactive variables, still requires primitive operations to be lifted. KScript [11] follows a different approach which allows regular object fields to be event streams. Additionally, KScript uses a pull-based update model and resolves sources of streams at each update which enables dynamic reconfiguration.

Instead of separating functional reactive programming from imperative programming, it is also possible to loosen the strict sequential execution of an otherwise imperative program with mutable state such that statements are automatically reordered and re-executed in reaction to changes [6, 9].

Finally, FRP, which propagates changes unidirectionally, can be generalized to constraint programming. When relations between time-varying reactive values are expressed as constraints, updates can propagate bidirectionally. Babelsberg/JS [8] is an example of an object constraint programming languages which can be executed by a standard JavaScript runtime. Different constraint solvers usually have different trade-offs in terms of performance and expressiveness, so Babelsberg/JS can use multiple different solvers depending on the concrete domain.

## 7. Discussion

Reactive variable declarations as presented in this paper offer an alternative approach to reactive programming that does not allow reification of signals. One the one hand, this simplifies reasoning about dependencies, which then have to follow scoping rules, and avoids the need to lift primitive computations. On the other hand, more research is necessary to evaluate whether this approach is practical for larger applications. In particular, it is not easily possible to provide generic functionality, e.g. *filter* or *pausable*, as a library function because reactive variables cannot be passed around as values. Additional, reactive variables and their dependencies cannot be dynamically reconfigured at runtime which limits the system to first-order reactive programming.

The static mapping of reactive dependencies to lexical scopes might be promising for future work on debugging and development tools that provide better visualizations and feedback for reactive programming. Furthermore, it would be possible to support asynchronous updates of reactive variables and a second type of reactive variable with continuous semantics that does not propagate updates with unchanged values.

## References


[1] 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. ISSN 0360-0300. doi: 10.1145/2501654.2501666.

[2] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33095-X, 978-3-540-33095-0. doi: 10.1007/11693024_20.

[3] A. Courtney. Frappé: Functional reactive programming in Java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL'01, pages 29–44, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41768-0.

[4] 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. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462161.

[5] T. Disney, N. Faubion, D. Herman, and C. Flanagan. Sweeten Your JavaScript: Hygienic Macros for ES5. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 35–44, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3211-8.

[6] 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. ISBN 978-1-60558-768-4. doi: 10.1145/1639950.1640058.

[7] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM. ISBN 0-89791-918-1. doi: 10.1145/258948.258973.

[8] T. Felgentreff, A. Borning, R. Hirschfeld, J. Lincke, Y. Ohshima, B. Freudenberg, and R. Krahn. Babelsberg/js. In *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 411–436. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44201-2.

[9] 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. ISBN 978-1-4503-3210-1. doi: 10.1145/2661136.2661145.

[10] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640091.

[11] 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. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509590.

[12] 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. ISBN 978-1-4503-2772-5. doi: 10.1145/2577080.2577083.

[13] T. Van Cutsem and M. Miller. Trustworthy proxies. In *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 154–178. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39037-1.

[14] T. Van Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0405-4. doi: 10.1145/1869631.1869638.