11-11-2022

Semantics of RxJS

Tian Zhao

Yonglun Li

Semantics of RxJS

Tian Zhao
University of Wisconsin – Milwaukee
Milwaukee, Wisconsin, USA
tzhao@uwm.edu

Yonglun Li
University of Wisconsin – Milwaukee
Milwaukee, Wisconsin, USA
yli@uwm.edu

Abstract

RxJS is a popular JavaScript library for reactive programming in Web applications. It provides numerous operators to create, combine, transform, and filter discrete events and to handle errors. These operators may be stateful and have side effects, which makes it difficult to understand the precise meaning of the resulting computation. In this paper, we define a formal model for RxJS programs by formalizing a selected subset of RxJS operators using a small-step operational semantics. We present several debugging related applications using the semantics as a model. We also implemented a subset of RxJS based on this semantics, which provides convenient access to the runtime representation of the RxJS program to help debugging.

CCS Concepts: • **Computing methodologies** → **Concurrent computing methodologies**; • **Software and its engineering** → **Concurrent programming structures**.

Keywords: Reactive Programming, RxJS, JavaScript

ACM Reference Format:
Tian Zhao and Yonglun Li. 2022. Semantics of RxJS. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '22), December 07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3563837.3568340

1 Introduction

RxJS is a reactive programming library for JavaScript, which has been integrated in frameworks such as Angular and React to handle UI and other asynchronous events. Despite its success, debugging of RxJS programs is still difficult. A recent study [1] examined the challenges in debugging RxJS programs due to the disparity between the declarative interface for defining dataflow logic [2] and the imperative implementation based on the Observer Pattern [13]. As pointed out in [1], traditional debugger does not offer much help in identifying the cause of the bugs since the dataflow logic of RxJS programs is not directly reflected in the control flow logic represented by the call stacks at the break points. A RxJS program constructs and mutates dataflow graphs and pushes events through the graphs to perform pure computation and to produce side effects. The dataflow graphs are implicit constructs that are not readily accessible to users during the debugging phase. To find the cause of an error, one needs to examine the states of the dataflow graphs, which include the graph shapes and the events at each graph node. Programmers often resort to indirect approaches such as printing event traces and drawing dataflow graphs by instrumenting the source program or using debugging tools such as rxjs-spy, rxviz, and rxjs-playground. While these methods are useful, they are informal, require manual inspection, and does not scale to larger programs.

In this paper, we present a formal semantics for a selected subset of RxJS operators to provide a precise definition of their meaning. Using the semantics as a model, we can define some debugging related applications to help discover problems in RxJS programs. The semantics models RxJS runtime with a heap that contains the subscription graph, where each node is a subscription, and a queue that contains the external and internal events. The reduction of a RxJS program alternates between graph construction phase and reactive phase where events are propagated through the graph. Using the semantics as a model, we can define a representation of stack trace specific to the RxJS programs, a set of rules to check subscription states, and a runtime invariant of the subscription graph to ensure that the error for each event source can be uniquely identified.

Based on this semantics, we have implemented a large subset of RxJS using a thread-like abstraction using concurrency monad [27, 28], where each thread can be cancelled via its thread ID. We use this thread abstraction to implement the subscription to an observable so that unsubscribe operation is the same as thread cancellation. When an observable is subscribed, it returns a subscription that holds an emitter, a thread ID, and the subscriptions to the source and child observables if any. Through the subscription object, users can cancel the subscription or use it to navigate the dataflow graphs, check the state of each node, and examine the previous events.

In this paper, we make the following contributions:

1. We give a motivational example in Section 2 to demonstrate how semantics can help identify problems in a RxJS program.

REBLS '22, December 07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9911-1/22/12.
https://doi.org/10.1145/3563837.3568340

2. In Section 3, we present a formal model of RxJS by defining an operational semantics for a selected subset of RxJS operators. The semantics provides a simplified model for reasoning about the expected and unexpected behavior of a RxJS program.
3. In Section 4, we provide some example applications of the semantics for debugging purposes, which include stack-trace suitable for reactive programs, unexpected states of observables, and checking invariant on subscription graph,
4. We describe an implementation of RxJS library that conforms to this semantics in Section 5.
5. The related works are discussed in Section 6.

2 Subscription as Dataflow Graph

In this section, we use the following example to motivate the need of a formal semantics for RxJS. This example is a simple type-ahead client, which sends a query request each time the user types a character in a text box. If a previous query does not complete before a new request, then old query is canceled. If a query is answered on time, then the result is displayed.

```
let subscription =
  fromEvent('#type-ahead', 'keyup') // input
  .map(e => e.target.value)     // input text
  .switchMap(x => from(query(x)))   // query
  .subscribe(display);     // display results
```

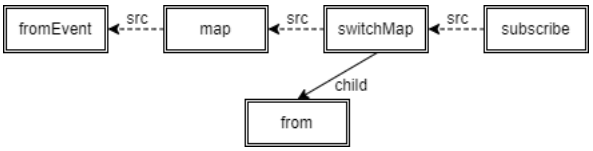

Figure 1. The subscription graph of the type-ahead example.

The structure of the subscription graph is shown in Figure 1, where the arrows point to the source or child of each subscription. The subscription graph mirrors the dataflow, from which we can also examine the past events emitted by each observable. There are three types of events emitted by an observable.

1. `next(v)`, which holds event value v;
2. `end`, which indicates that the observable completes;
3. `error`, which signals an exception and it recursively propagates to the subscribers until it is either caught by a `catchError` observable or by the root subscriber. Observable that throws an error is cancelled.

Using the subscription graph, users can find the source of errors more easily. For example, the type-ahead server returns an error for certain inputs. The user wants to catch the error by inserting a `catchError` operator so that the error does not stop the client.

```
let subscription =
  fromEvent('#type-ahead', 'keyup')
  .map(e => e.target.value)
  .switchMap(x => from(query(x)))
  .catchError(_ => of('error'))  // A bug
  .subscribe(display);
```

The code above shows the initial attempt of catching error but it fails to keep the type-ahead client running – it still terminates if an error is emitted. The cause of the problem is that the `catchError` protects the `switchMap`, and when it catches an error, it must terminate the entire `switchMap` instead of the `from` observable.

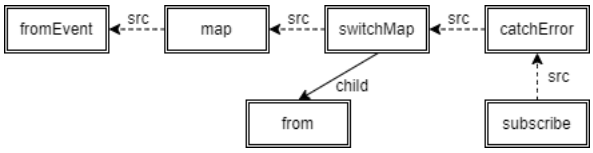

Figure 2. The initial attempt to catch query errors.

This relation is illustrated in Figure 2. Note that while it is possible to spot this problem by inspecting the source code, it is much harder to identify similar problems in larger applications. A systematic solution should allow automatic analysis of the subscription graph to detect similar problems. For example, we can require that every source observable such as `fromEvent` or `from` be uniquely protected by a `catchError` or a root subscriber. In Figure 2, both the `fromEvent` and `from` are guarded by the same `catchError`, which may indicate a potential problem.

We can fix the bug by placing the `catchError` within the `switchMap` operator as shown below.

```
let subscription =
  fromEvent('#type-ahead', 'keyup')
  .map(e => e.target.value)
  .switchMap(x =>
       from(query(x))
       .catchError(_ => of('error'))
   ) // catch error and emit a string event
  .subscribe(display);
```

The resulting subscription graph is shown in Figure 3, where `fromEvent` is protected by the root subscriber while `from` is protected by `catchError`.

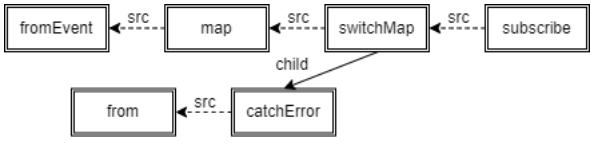

Figure 3. The correct handling of query errors.

While the subscription graph is static in this example, in general, a subscription graph may change at runtime. This

means that the checking of the subscription graph should be done at runtime when the graph is updated. However, to avoid unnecessary overhead, the runtime checking should be limited to graph updates that may cause a violation of the invariant. To understand how to check this invariant, in the next section, we present a formal semantics to a selected set of RxJS operators.

3 Operational Semantics

In this section, we define a formal model of RxJS computation using a selected subset of RxJS operators, which we call λ_{rx}. We define an operational semantics that includes the rules for subscribe and unsubscribe operations and the rules for reacting to external or internal events.

3.1 Syntax

The syntax of λ_{rx} is shown in Figure 4, where e ranges over expressions such as variables, observables, subscribe operations $e.\texttt{sub}(f)$, unsubscribe operations $e.\texttt{unsub}()$, assignments, sequences, functions, and calls. The subscribe operation $e.\texttt{sub}(f)$ starts an observable e and calls a function f with the observable events as inputs. The unsubscribe operation stops the subscription to an observable. Note that $o.\texttt{sub}()$ is a runtime value that denotes the subscription to the observable o and it evaluates to a reference r to the subscription. The assignment expression $x = o.\texttt{share}()$ is used to share the observable o through the variable x.

The variable o ranges over x (for shared observables) and observable expressions, which include one or two operators from each group of RxJS operators detailed as follows.

- $\texttt{fromEvent}(elm, evt)$ is a creation operator that creates a primitive observable that emits events of type evt from a DOM element elm.
- $\texttt{combine}(o_1, o_2)$ (the same as combineLatest in RxJS) and $\texttt{concat}(o_1, o_2)$ are combination operators that combine the latest events from o_1 and o_2 and concatenate the event streams of o_1 and o_2.
- $o.\texttt{every}(f)$ is a condition operator that emits true if all events from o satisfy the predicate f and emits false otherwise.
- $o.\texttt{catchError}(f)$ is an error handling operator that emits next events from o but if o emits an error, then it stops o and emits events from $f()$ instead.
- $o.\texttt{share}()$ is a multicasting operator that shares the events of o with multiple subscribers and manages the lifetime of o using reference counting. When the shared observable is first subscribed, it is transformed to $\texttt{share}(o, r)$, where r points to a subscription of o.
- $o.\texttt{take}(n)$ is a filtering operator that takes up to n events from o.
- $o.\texttt{map}(f)$ is a transformation operator that for each event value v of o, it emits $f(v)$.

$$
\begin{array}{lll}
e \in \text{Expression} & ::= & x \\
& | & o \\
& | & e.\texttt{sub}(f) \\
& | & e.\texttt{unsub}() \\
& | & \colorbox{gray!15}{$o.\texttt{sub}()$} \\
& | & x = o.\texttt{share}() \\
& | & e;\ e' \\
& | & x \Rightarrow e \\
& | & e(e') \\
& | & \ldots \\
o \in \text{Observable} & ::= & x \\
& | & \texttt{fromEvent}(elm, evt) \\
& | & \texttt{combine}(o_1, o_2) \\
& | & \texttt{concat}(o_1, o_2) \\
& | & o.\texttt{every}(f) \\
& | & o.\texttt{catchError}(f) \\
& | & o.\texttt{take}(n) \\
& | & o.\texttt{map}(f) \\
& | & o.\texttt{switchMap}(f)
\end{array}
$$

Figure 4. The syntax of λ_{rx}, where the shaded terms are runtime entities.

- $o.\texttt{switchMap}(f)$ is also a transformation operator that for each event value v of o, it emits the events of the new inner observable $f(v)$ after it stops the existing inner observable if any.

3.2 Runtime Values

A subscribe operation $o.\texttt{sub}()$ evaluates to a subscription, which is a runtime value shown in Figure 5. Each observable operator has a corresponding subscription value, which may carry additional state information of the subscription. We store the subscriptions in a heap H that maps references (denoted by r) to subscriptions. Each subscription can refer to its source subscriptions through the references, which is detailed as follows.

- $\texttt{combine}(r_1, r_2, v_1, v_2, n)$ contains r_1 and r_2, which are the subscriptions to the source observables. It also contains v_1 and v_2, which are either the last events or undefined (denoted by ϵ). n is the number of source observables that have completed.
- $\texttt{concat}(r_1, o_2)$ contains the subscription r_1 and an observable expression o_2 that starts once r_1 completes.
- $\texttt{every}(r, f)$, $\texttt{catchError}(r, f)$, and $\texttt{map}(r, f)$ contain the subscription r to their source observable.
- $\texttt{sharing}(x)$ contains x that references a shared observable $\texttt{share}(o, r)$, where r is a subscription to o.
- $\texttt{take}(r, n)$ contains the subscription r to its source observable and the remaining number of events n.

$$
\begin{array}{lcl}
s \in \text{Subscription} & ::= & \texttt{fromEvent}(elm, evt) \\
& | & \texttt{combine}(r_1, r_2, v_1, v_2, n) \\
& | & \texttt{concat}(r_1, o_2) \\
& | & \texttt{every}(r, f) \\
& | & \texttt{catchError}(r, f) \\
& | & \texttt{sharing}(x) \\
& | & \texttt{take}(r, n) \\
& | & \texttt{map}(r, f) \\
& | & \texttt{switchMap}(r_1, f, r_2, b) \\
& | & \texttt{sub}(r, f) \\
v \in \text{Value} & ::= & n \mid o \mid \text{true} \mid \text{false} \mid \epsilon \mid x \Rightarrow e \\
h \in \text{Shared observable} & ::= & o.\texttt{share}() \\
& | & \texttt{share}(o, r) \\
H \in \text{Heap} & ::= & \{x \mapsto h; \ldots; r \mapsto s; \ldots\} \\
Q \in \text{Queue} & ::= & [(t, r), \ldots] \\
t \in \text{Event} & :: & \texttt{next}(v) \\
& | & \text{end} \\
& | & \text{error}
\end{array}
$$

Figure 5. The runtime values of λ_{rx}, where x points to shared observables and r points to the subscriptions in the heap.

- $\texttt{switchMap}(r_1, f, r_2, b)$ contains the subscriptions to the outer and inner observables r_1, r_2, and b, which is a Boolean that indicates whether the inner or outer observable has ended.

A queue Q is used to temporarily hold events that are emitted from observables but have not yet been received by the subscribers. Q holds a list of pairs (t, r), where t is an event and r refers to the subscription that emitted the event. Each event is either a $\texttt{next}(v)$ event with value v, an error, or an end event that signals the completion of an observable. The event queue can be used to model some variations of event scheduling in RxJS.

3.3 Evaluation of Expressions

λ_{rx} includes variables, observables, assignments, functions, calls, and subscribe and unsubscribe operations. The evaluation rules for expressions are shown in Figure 7, where each rule has the form of $H, e \longrightarrow H', e'$. The expressions such as assignment and unsubscribe operation are evaluated for their side effects.

The expression $o.\texttt{sub}(f)$ is the starting point of RxJS computation, where the observable o is subscribed so that its events are used to run f for its side effects. By Rule E_{sub1}, the subscribe operation evaluates to a reference r that points to $\texttt{sub}(r', f)$, which listens on the events of r' to run f and r' references the subscription to o. By Rule E_{sub2}, $o.\texttt{sub}()$ reduces to a fresh reference that maps to the subscription

$$
\begin{array}{lcl}
E & ::= & \cdot \\
& | & E.\texttt{sub}(f) \\
& | & E.\texttt{unsub}() \\
& | & E; e \\
& | & E(e) \\
& | & v(E)
\end{array}
$$

Figure 6. The evaluation context for expressions

$$
\begin{array}{cl}
H, v \longrightarrow H, v & E_{Val} \\
\dfrac{r \text{ is fresh} \quad H, o.\texttt{sub}() \longrightarrow H', r'}{H, o.\texttt{sub}(f) \longrightarrow H'[r \mapsto \texttt{sub}(r', f)], r} & E_{Sub1} \\
\dfrac{r \text{ is fresh} \quad H, o.\texttt{sub}() \rightsquigarrow H', s}{H, o.\texttt{sub}() \longrightarrow H'[r \mapsto s], r} & E_{Sub2} \\
H, x = o.\texttt{share}() \longrightarrow H[x \mapsto o.\texttt{share}()], \epsilon & E_{Assign} \\
\dfrac{H, r.\texttt{unsub}() \rightsquigarrow H'}{H, r.\texttt{unsub}() \longrightarrow H' \backslash \{r \mapsto H'(r)\}, \epsilon} & E_{Unsub} \\
\dfrac{H, e \longrightarrow H', e'}{H, E[e] \longrightarrow H', E[e']} & E_{Cong} \\
H, v; e \longrightarrow H, e & E_{Seq} \\
H, (x \Rightarrow e)(v) \longrightarrow H, [v/x]e & E_{Call}
\end{array}
$$

Figure 7. The evaluation rules for expressions

value s evaluated from $o.\texttt{sub}()$. The evaluation of $o.\texttt{sub}()$ to a subscription value is defined by the rules in Figure 8.

Subscribe Operations. In λ_{rx}, each subscribe operation reduces to a reference r that is mapped to a subscription value s in the heap. The subscribers to s receives its events through r such that the subscriptions are linked as a graph through references like r.

A subscribe operation $o.\texttt{sub}()$ may cause additional subscriptions to the observables in o. This behavior is detailed in Figure 8. For example, in Rule $S_{combine}$, the subscription to a '`combine`' observable leads to the subscriptions to its source observables o_1 and o_2.

Most of the RxJS operators create *cold* observables [20], each of which has a single subscriber. The group of multicast operators such as `share` create *hot* observables that can be shared by multiple subscribers. Rule S_{Share1} says that when $H(x) = o.\texttt{share}()$ is subscribed the first time in a call $x.\texttt{sub}()$, o is subscribed and the heap H is updated so that x is mapped to $\texttt{share}(o, r)$, where r points to the subscription to o. By Rule S_{Share2}, each subsequent call to $x.\texttt{sub}()$ reduces to $\texttt{sharing}(x)$, which is a subscription value that represents shared access to $\texttt{share}(o, r)$.

$$\frac{o = \mathsf{fromEvent}(elm, evt)}{H, o.\mathsf{sub}() \rightsquigarrow H, \mathsf{fromEvent}(elm, evt)} \; S_{From}$$

$$\frac{\begin{array}{c} o = \mathsf{combine}(o_1, o_2) \\ H, o_1.\mathsf{sub}() \rightarrow H_1, r_1 \\ H_1, o_2.\mathsf{sub}() \rightarrow H_2, r_2 \end{array}}{H, o.\mathsf{sub}() \rightsquigarrow H_2, \mathsf{combine}(r_1, r_2, \epsilon, \epsilon, 0)} \; S_{Combine}$$

$$\frac{\begin{array}{c} o = H, \mathsf{concat}(o_1, o_2) \\ H, o_1.\mathsf{sub}() \rightarrow H_1, r_1 \end{array}}{H, o.\mathsf{sub}() \rightsquigarrow H_1, \mathsf{concat}(r_1, o_2)} \; S_{Concat}$$

$$\frac{o = o'.\mathsf{every}(f) \quad H, o'.\mathsf{sub}() \rightarrow H', r}{H, o.\mathsf{sub}() \rightsquigarrow H', \mathsf{every}(r, f)} \; S_{Every}$$

$$\frac{\begin{array}{c} o = o'.\mathsf{catchError}(f) \\ H, o'.\mathsf{sub}() \rightarrow H', r \end{array}}{H, o.\mathsf{sub}() \rightsquigarrow H', \mathsf{catchError}(r, f)} \; S_{Catch}$$

$$\frac{\begin{array}{c} H(x) = o.\mathsf{share}() \\ H, o.\mathsf{sub}() \rightarrow H', r \\ H'' = H'[x \mapsto \mathsf{share}(o, r)] \end{array}}{H, x.\mathsf{sub}() \rightsquigarrow H'', \mathsf{sharing}(x)} \; S_{Share1}$$

$$\frac{H(x) = \mathsf{share}(o, r)}{H, x.\mathsf{sub}() \rightsquigarrow H, \mathsf{sharing}(x)} \; S_{Share2}$$

$$\frac{o = o'.\mathsf{take}(n) \quad H, o'.\mathsf{sub}() \rightarrow H', r}{H, o.\mathsf{sub}() \rightsquigarrow H', \mathsf{take}(r, n)} \; S_{Take}$$

$$\frac{o = o'.\mathsf{map}(f) \quad H, o'.\mathsf{sub}() \rightarrow H', r}{H, o.\mathsf{sub}() \rightsquigarrow H', \mathsf{map}(r, f)} \; S_{Map}$$

$$\frac{\begin{array}{c} o = o'.\mathsf{switchMap}(f) \\ H, o'.\mathsf{sub}() \rightarrow H', r \end{array}}{H, o.\mathsf{sub}() \rightsquigarrow H', \mathsf{switchMap}(r, f, \epsilon, \mathsf{false})} \; S_{Switch}$$

Figure 8. The rules for subscribe operations.

$$\frac{H(r) = \mathsf{fromEvent}(elm, evt)}{H, r.\mathsf{unsub}() \rightsquigarrow H} \; U_{From}$$

$$\frac{\begin{array}{c} H(r) = \mathsf{combine}(r_1, r_2, v_1, v_2, n) \\ H, r_1.\mathsf{unsub}() \rightarrow H_1 \\ H_1, r_2.\mathsf{unsub}() \rightarrow H_2 \end{array}}{H, r.\mathsf{unsub}() \rightsquigarrow H_2} \; U_{Combine}$$

$$\frac{\begin{array}{c} H(r) = \mathsf{concat}(r_1, o_2) \\ H, r_1.\mathsf{unsub}() \rightarrow H_1 \end{array}}{H, r.\mathsf{unsub}() \rightsquigarrow H_1} \; U_{Concat}$$

$$\frac{H(r) = \mathsf{every}(r', f) \quad H, r'.\mathsf{unsub}() \rightarrow H'}{H, r.\mathsf{unsub}() \rightsquigarrow H'} \; U_{Every}$$

$$\frac{\begin{array}{c} H(r) = \mathsf{catchError}(r', f) \\ H, r'.\mathsf{unsub}() \rightarrow H' \end{array}}{H, r.\mathsf{unsub}() \rightsquigarrow H'} \; U_{Catch}$$

$$\frac{\begin{array}{c} H(r) = \mathsf{sharing}(x) \\ \exists r' \neq r.\ H(r') = \mathsf{sharing}(x) \end{array}}{H, r.\mathsf{unsub}() \rightsquigarrow H} \; U_{Share1}$$

$$\frac{\begin{array}{c} H(r) = \mathsf{sharing}(x) \quad H(x) = \mathsf{share}(o, r') \\ \nexists r'' \neq r.\ H(r'') = \mathsf{sharing}(x) \\ H, r'.\mathsf{unsub}() \rightarrow H' \end{array}}{H, r.\mathsf{unsub}() \rightsquigarrow H'[x \mapsto o.\mathsf{share}()]} \; U_{Share2}$$

$$\frac{H(r) = \mathsf{take}(r', n) \quad H, r'.\mathsf{unsub}() \rightarrow H'}{H, r.\mathsf{unsub}() \rightsquigarrow H'} \; U_{Take}$$

$$\frac{H(r) = \mathsf{map}(r', f) \quad H, r'.\mathsf{unsub}() \rightarrow H'}{H, r.\mathsf{unsub}() \rightsquigarrow H'} \; U_{Map}$$

$$\frac{\begin{array}{c} H(x) = \mathsf{switchMap}(r_1, f, r_2, b) \\ H, r_1.\mathsf{unsub}() \rightarrow H_1 \\ H_1, r_2.\mathsf{unsub}() \rightarrow H_2 \end{array}}{H, r.\mathsf{unsub}() \rightsquigarrow H_2} \; U_{Switch}$$

$$\frac{H(r) = \mathsf{sub}(r', f) \quad H, r'.\mathsf{unsub}() \rightarrow H'}{H, r.\mathsf{unsub}() \rightsquigarrow H'} \; U_{Sub}$$

Figure 9. The rules for unsubscribe operations.

Unsubscribe Operations. While the subscribe operation constructs the subscription graph from the observables, the unsubscribe operation does the opposite by removing the subscriptions to the observables from the heap. Rule E_{Unsub} in Figure 7 evaluates $r.\mathsf{unsub}()$ by first applying the rules for unsubscribe operation in Figure 9 and then removing the mapping of r from the heap. The rules in Figure 9 apply the E_{Unsub} rule to the references in each subscription value to unsubscribe them recursively.

The only exceptions are the rules for the shared observable, which uses reference counting to decide whether to unsubscribe its source. By Rule U_{Share1}, if a shared observable x is still used by another subscription r', then the unsubscribe operation $r.\mathsf{unsub}()$ does not change the heap. Otherwise, by Rule U_{Share2}, the source observable of r will be unsubscribed.

3.4 Event Propagation

Once a subscription graph is constructed, the next stage of computation is event propagation, which may be interleaved with further modification to the subscription graph. The event propagation computation is described by the rules in Figure 10 and 11, where each reduction step has the form of $H, Q \rightarrow H', Q'$. Each rule (except Rule R_{From}) removes an event and subscription pair (t, r_1) from the front of the event queue Q and sends t to r_1's subscriber(s) – r, which may trigger further updates to the heap and the queue.

Most of the rules are concerned with $\mathsf{next}(v)$ event and end event. By Rule R_{Error}, the error event from a reference

r is forwarded by default to the subscriber of r. This rule applies to the error propagation of all observables except the shared observable, which is handled by Rule R_{Share}

By Rule R_{Catch}, the error event of r' can be handled by the subscription $H(r) = \texttt{catchError}(r', f)$, which calls the error handler f to obtain an observable o, subscribes to o, and maps r to the subscription to o. In the end, r' is unsubscribed.

When processing events in the subscription graph, the new events may be added to the back or the front of the queue. For example, by Rule R_{From}, the event emitted from the subscription to $\texttt{fromEvent}(elm, evt)$ is added to the back of the event queue. This is consistent with the event handling of JavaScript, where asynchronous events are processed in the macro task queue.

Other than external events such as those from DOM elements, there are also internal events created by RxJS operators. For example, the `combine` operator emits the latest values from its sources when one of them emits (after both have emitted). By Rule $R_{Combine2}$, the event of the `combine` operator is put in front of the queue so that it is received immediately by the subscriber of this observable. Rule $R_{Combine1}$ applies to the case when one of the source observable has not emitted any event. By Rules $R_{Combine3}$ and $R_{Combine4}$, an end event is emitted after all source observables have completed. The end event is also placed in front of the queue so that it is received immediately by the subscriber.

By Rules $R_{Concat1}$ and $R_{Concat2}$, $\texttt{concat}(o_1, o_2)$ combines the event streams of o_1 and o_2 sequentially such that it subscribes to o_1 first and after o_1 completes, it subscribes to o_2. The observable $o.\texttt{every}(f)$ emits a false value immediately if an event value from o does not satisfy the predicate function f (by Rule R_{Every1}). It emits a true value after all its events satisfy f (by Rules R_{Every2} and R_{Every3}).

By Rule R_{Share}, the observable $o.\texttt{share}()$ broadcasts the events of o to all subscribers of the shared observable. The observable $o.\texttt{take}(n)$ forwards the first n events of o to its subscriber (by Rules R_{Take1} and R_{Take2}) or until o emits an end event (by Rule R_{Take3}). Note that if the subscription r' of o is still running after n events have been emitted, it must be unsubscribed (Rule R_{Take2}).

By Rules R_{Map1} and R_{Map2}, the observable $o.\texttt{map}(f)$ transforms each $\texttt{next}(v)$ event from o and re-emits the result. For $o.\texttt{switchMap}(f)$, there are 5 rules: Rule $R_{Switch1}$ says that the events from the inner observable are re-emitted. Rules $R_{Switch2}$ and $R_{Switch3}$ describe how the event $\texttt{next}(v)$ from the outer observable interrupts the inner observable (if any) and starts a new subscription to the observable evaluated from $f(v)$. Rules $R_{Switch4}$ and $R_{Switch5}$ say that if both of the outer and inner observables have completed, then an end event is emitted.

Rules R_{Sub1}, R_{Sub2}, and R_{Sub3} define the behavior of a top-level subscription $o.\texttt{sub}(f)$, which calls f with v for each $\texttt{next}(v)$ event from o until an end event is emitted. It stops the subscription to o if an error is emitted. Note that in RxJS, separate callback functions can be given to handle the end and the error events but this detail is omitted for simplicity.

$$\frac{H(x) = \texttt{fromEvent}(elm, evt) \quad elm \text{ emits } evt}{H, Q \to H, Q@[(\texttt{next}(evt), x)]} \; R_{From}$$

$$\frac{H(r) = \texttt{combine}(r_i, r_j, v_i, \epsilon, n) \quad i \neq j \quad H' = H[r \mapsto \texttt{combine}(r_i, r_j, v, \epsilon, n)]}{H, (\texttt{next}(v), r_i) :: Q \to H', Q} \; R_{Combine1}$$

$$\frac{H(r) = \texttt{combine}(r_i, r_j, v_i, v_j, n) \quad i \neq j \quad H' = H[r \mapsto \texttt{combine}(r_i, r_j, v, v_j, n)] \quad Q' = (\texttt{next}((v, v_j)), r) :: Q}{H, (\texttt{next}(v), r_i) :: Q \to H', Q'} \; R_{Combine2}$$

$$\frac{H(r) = \texttt{combine}(r_i, r_j, v_i, v_j, 0) \quad i \neq j \quad H' = H[r \mapsto \texttt{combine}(r_i, r_j, v_i, v_j, 1)]}{H, (\texttt{end}, r_i) :: Q \to H', Q} \; R_{Combine3}$$

$$\frac{H(r) = \texttt{combine}(r_i, r_j, v_i, v_j, 1) \quad i \neq j}{H, (\texttt{end}, r_i) :: Q \to H, (\texttt{end}, r) :: Q} \; R_{Combine4}$$

$$\frac{H(r) = \texttt{concat}(r_1, o_2)}{H, (\texttt{next}(v), r_1) :: Q \to H, (\texttt{next}(v), r) :: Q} \; R_{Concat1}$$

$$\frac{H(r) = \texttt{concat}(r_1, o_2) \quad H, o_2.\texttt{sub}() \rightsquigarrow H', s \quad H'' = H'[r \mapsto s]}{H, (\texttt{end}, r_1) :: Q \to H'', Q} \; R_{Concat2}$$

$$\frac{H(r) = \texttt{every}(r_1, f) \quad H, f(v) \to H', \texttt{false} \quad H', r_1.\texttt{unsub}() \to H'' \quad Q' = [(\texttt{next}(\texttt{false}), r), (\texttt{end}, r)]@Q}{H, (\texttt{next}(v), r_1) :: Q \to H'', Q'} \; R_{Every1}$$

$$\frac{H(r) = \texttt{every}(r_1, f) \quad H, f(v) \to H', \texttt{true}}{H, (\texttt{next}(v), r_1) :: Q \to H', Q} \; R_{Every2}$$

$$\frac{H(r) = \texttt{every}(r_1, f) \quad Q' = [(\texttt{next}(\texttt{true}), r), (\texttt{end}, r)]@Q}{H, (\texttt{end}, r_1) :: Q \to H, Q'} \; R_{Every3}$$

$$\frac{H(r) = \texttt{catchError}(r_1, f) \quad H, f() \to H_1, o \quad H_1, o.\texttt{sub}() \rightsquigarrow H_2, s \quad H_2[r \mapsto s], r_1.\texttt{unsub}() \to H'}{H, (\texttt{error}, r_1) :: Q \to H', Q} \; R_{Catch}$$

$$\frac{r_1 \text{ appears in } H(r)}{H, (\texttt{error}, r_1) :: Q \to H, (\texttt{error}, r) :: Q} \; R_{Error}$$

Figure 10. The reduction rules for subscriptions 1

Scheduler and Event Queue. RxJS supports multiple types of event schedulers. By default, internal events are scheduled synchronously. For example, the events of the

$$\frac{\exists x.\, H(x) = \mathsf{share}(o, r) \quad \forall i \in \{1..n\}.\, H(r_i) = \mathsf{sharing}(x) \quad Q' = [(t, r_1), \ldots, (t, r_n)]@Q}{H, (t, r) :: Q \rightarrow H, Q'} \; R_{Share}$$

$$\frac{H(r) = \mathsf{take}(r', n) \quad n \geq 2 \quad H' = H[r \mapsto \mathsf{take}(r', n-1)]}{H, (t, r') :: Q \rightarrow H', (t, r) :: Q} \; R_{Take1}$$

$$\frac{H(r) = \mathsf{take}(r', 1) \quad H, r'.\mathsf{unsub}() \rightarrow H'}{H, (t, r') :: Q \rightarrow H', [(t, r), (\mathsf{end}, r)]@Q} \; R_{Take2}$$

$$\frac{H(r) = \mathsf{take}(r', n)}{H, (\mathsf{end}, r') :: Q \rightarrow H, (\mathsf{end}, r) :: Q} \; R_{Take3}$$

$$\frac{H(r) = \mathsf{map}(r', f) \quad H, f(v) \rightarrow H', v'}{H, (\mathsf{next}(v), r') :: Q \rightarrow H', (\mathsf{next}(v'), r) :: Q} \; R_{Map1}$$

$$\frac{H(r) = \mathsf{map}(r', f)}{H, (\mathsf{end}, r') :: Q \rightarrow H, (\mathsf{end}, r) :: Q} \; R_{Map2}$$

$$\frac{H(r) = \mathsf{switchMap}(r_1, f, r_2, b)}{H, (\mathsf{next}(v), r_2) :: Q \rightarrow H, (\mathsf{next}(v), r) :: Q} \; R_{Switch1}$$

$$\frac{H(r) = \mathsf{switchMap}(r_1, f, \epsilon, b) \quad H, f(v) \rightarrow H_1, o \quad H_1, o.\mathsf{sub}() \rightarrow H_2, r_2 \quad H_3 = H_2[r \mapsto \mathsf{switchMap}(r_1, f, r_2, b)]}{H, (\mathsf{next}(v), r_1) :: Q \rightarrow H_3, Q} \; R_{Switch2}$$

$$\frac{H(r) = \mathsf{switchMap}(r_1, f, r_2, b) \quad H, f(v) \rightarrow H_1, o \quad H_1, o.\mathsf{sub}() \rightarrow H_2, r_2' \quad H_3 = H_2[r \mapsto \mathsf{switchMap}(r_1, f, r_2', \mathsf{false})] \quad H_3, r_2.\mathsf{unsub}() \rightarrow H'}{H, (\mathsf{next}(v), r_1) :: Q \rightarrow H', Q} \; R_{Switch3}$$

$$\frac{H(r) = \mathsf{switchMap}(r_1, f, r_2, \mathsf{true}) \quad Q' = (\mathsf{end}, r) :: Q}{H, (\mathsf{end}, r_i) :: Q \rightarrow H, Q' \quad i \in \{1, 2\}} \; R_{Switch4}$$

$$\frac{H(r) = \mathsf{switchMap}(r_1, f, r_2, \mathsf{false}) \quad H' = H[r \mapsto \mathsf{switchMap}(r_1, f, r_2, \mathsf{true})]}{H, (\mathsf{end}, r_i) :: Q \rightarrow H', Q \quad i \in \{1, 2\}} \; R_{Switch5}$$

$$\frac{H(r) = \mathsf{sub}(r', f) \quad f(v)}{H, (\mathsf{next}(v), r') :: Q \rightarrow H, Q} \; R_{Sub1}$$

$$\frac{H(r) = \mathsf{sub}(r', f)}{H, (\mathsf{end}, r') :: Q \rightarrow H, Q} \; R_{Sub2}$$

$$\frac{H(r) = \mathsf{sub}(r', f) \quad H, r'.\mathsf{unsub}() \rightsquigarrow H'}{H, (\mathsf{error}, r') :: Q \rightarrow H', Q} \; R_{Sub3}$$

Figure 11. The reduction rules for subscriptions 2

`combine` operator can be passed to its subscribers directly via function calls. However, a subscription can run on schedulers such as the async scheduler, which is based on the JavaScript queue for asynchronous events. The rules in Figure 10 and 11 correspond to the default scheduler where the internal events are placed in front of the queue so that they must be processed immediately. To emulate the behavior of an async scheduler, we can modify the rules so that the internal events are placed at the end of Q so that they are processed after the events on the queue are processed.

3.5 Additional Operators

RxJS has over 100 operators and this semantics only considered a subset selected based on functionalities, which include combination, creation, error handling, multicasting, filtering, and transformation. Other operators in these classes can be formalized in similar ways with additional complexities such as buffering.

For example, the operator `o.concatAll()` subscribes to each of the observables emitted from `o` and concatenates the resulting event streams. This operator uses a buffer to hold the new observables emitted from o while waiting for the current observable to complete. The operator `zip(o1,o2)` pairs events from `o1` and `o2` until one of them completes and buffers are needed to hold events from either observables in case that they do not emit at the same rate.

Some operators may be implemented using others. For example, `o.combineAll()` combines the events from the observables emitted from `o` in tuples, which can be implemented using `combineLatest` after collecting all the observables from `o`. The operator `o.endsWith(x)` appends to the events from `o` with the value `x`, which can be implemented with `concat` and the operator `of` (that creates an observable out of values).

The semantics of some operators is difficult to characterize concisely. For example, the operator `o.debounce(selector)` controls the emission rate of `o` by racing its events with the observable returned from the `selector` function. Other filter operators such as `distinct` and `throttle` also have complex semantics related to the value and timing of the events. Even more complex are the transformation operators that group the source events using buffers or windows based on the event count, event timing, and timing observables.

Lastly, the multicasting operators use classes like `Subject` to broadcast events from a source observable to multiple subscribers. The operators such as `window` and `groupBy` can use `Subject` to implement observables that emit events selected from a source observable. The `Subject` class also has methods for direct event emission, which can be used to emit events for a observable from any part of the program as a side effect. While we can represent operators like `share`, we have not modeled this type of event emission.

4 Debugging with Subscription Graph

The semantics of λ_{rx} provides a simplified model for describing the debugging support for RxJS programs. In this

section, we discuss the representation of stack trace, detecting unexpected states of observables, and checking the invariant of subscription graph.

4.1 Stack Trace

For sequential programs, debuggers can be used to set break-points to pause the execution so that the programmers can examine the runtime states and discover the sources of errors. The runtime state includes a call stack with a list of stack frames that contains the local variables. The call stack can be dumped as a stack trace if a program crashes due to an exception. However, for RxJS program, such a stack trace is not very informative for debugging purposes. The dataflow information of a subscription graph is obscured by the underlying implementation of RxJS. For example, the event propagation in the subscription graph may appear as direct function calls, invocation of callback listeners, or resolution of promises depending on the types of the events and the event scheduler. Thus, we need a more abstract representation of the runtime state of a RxJS program.

The stack trace of a RxJS program may include the stack frames of normal function calls and/or the dataflow paths from observables to their subscribers. There are three types of stack traces in a RxJS program.

- The first type represents the construction phase of a subscription graph, where the stack trace consists of only stack frames.
- The second type represents the reactive phase of the computation, where the trace includes only dataflow paths. Since the only event source in λ_{rx} is asynchronous, event propagation will not start until the synchronous computation has completed, which means that the call stack is empty at this point.
- The third type includes dataflow paths followed by stack frames, which may occur when an expression break-point is reached during event propagation. For example, in $o = o'.\texttt{switchMap}(f)$, when the outer observable o' emits an event, the function f runs. If a break-point within f is reached, then the stack trace will include the dataflow path that leads to o and the call stack that leads to the break-point in f.

Break point. In λ_{rx}, we can set break-points for expressions by placing labels. For example, the expression e^{ℓ} has the break-point ℓ. When a computation reaches the break-point ℓ as shown below, it pauses with the current heap H and call stack E.

$$\frac{H_1, e_1 \rightarrow H_2, e_2}{H_1, E[e_1^{\ell}] \rightarrow H_2, E[e_2]}$$

We can set break-point for subscription s by attaching a label ℓ to the observable that s is reduced from.

$$H, o^{\ell}.\mathsf{sub}() \rightsquigarrow H', s^{\ell}$$

If an event is received at or emitted from a subscription s with a break-point ℓ, where $H(r) = s^{\ell}$ and $H, Q \rightarrow H', Q'$, then the program pauses and allows programmers to inspect the current state H' and Q'. This type of break-points can be enhanced with filters to limit the type of events.

Dataflow Path. The stack trace may include dataflow path if a break-point is reached at a subscription or at an expression that runs due to event propagation (e.g. `switchMap`). The dataflow path should allow a programmer to trace the event sources from the break-point. Since each subscription has references to its source subscriptions, we can follow these references to recover the dataflow path. However, if a subscription has multiple event sources (e.g. `combine`), then we must know the latest event source, which can be implemented with a flag. Detailed data such as the last k events and their time-stamps can be added to help with debugging.

4.2 Subscription State

We can debug a RxJS program by monitoring the state of a subscription. RxJS has numerous operators and the subscription to each operator have multiple states. If a subscription enters a state unexpected by the programmers, the resulting behavior may cause an error that is hard to debug.

For example, the subscription to a `combine` observable will emit an event only after both of its source observables have emitted at least one event. If one of the source observable completes before emitting any value, then the `combine` observable will not emit anything regardless how the other source observable behaves. Thus, it may be useful to raise an alert that one of the source observables to a `combine` observable only emits an `end` event. This may help identify the cause why a `combine` observable never emits.

For the subscription $\texttt{concat}(r_1, o_2)$, we can raise an alert if r_1 completes without emitting any value. This may be incorrect since the result of the concatenation would be entirely that of o_2, which may be unexpected.

For the subscription $\texttt{every}(r_1, f)$, if its source observable r_1 ends before firing any event, it will emit `true` and then complete instead of ending without emitting any value. This behavior may not be expected either.

For the subscription $\texttt{switchMap}(r_1, f, r_2, b)$, if the outer observable r_1 emits before the inner observable r_2 emits any event, then the `switchMap` observable will not fire any events from r_2. Furthermore, if r_1 always emits events at a higher rate than r_2, then the `switchMap` may not emit anything. Thus, we may want to raise an alert if an inner observable of `switchMap` is unsubscribed before it emits any event.

4.3 Runtime Invariant

When a RxJS program crashes, it is not always clear where the error comes from, especially when there are multiple event sources. An unhandled error from any of the event sources can cause the entire program to stop. One way to

$$\mathsf{safe}_H(\texttt{fromEvent}(\_,\_),1)$$

$$\frac{\mathsf{safe}_H(r_1,n_1) \quad \mathsf{safe}_H(r_2,n_2) \quad n_1+n_2 \le n}{\mathsf{safe}_H(\texttt{combine}(r_1,r_2,\_,\_,\_),n)}$$

$$\frac{\mathsf{safe}_H(r,n)}{\mathsf{safe}_H(\texttt{concat}(r,\_),n)} \qquad \frac{\mathsf{safe}_H(r,n)}{\mathsf{safe}_H(\texttt{every}(r,\_),n)}$$

$$\frac{\mathsf{safe}_H(r,1)}{\mathsf{safe}_H(\texttt{catchError}(r,\_),0)} \qquad \frac{\mathsf{safe}_H(r,n)}{\mathsf{safe}_H(\texttt{sharing}(r),n)}$$

$$\frac{\mathsf{safe}_H(r,n)}{\mathsf{safe}_H(\texttt{take}(r,\_),n)} \qquad \frac{\mathsf{safe}_H(r,n)}{\mathsf{safe}_H(\texttt{map}(r,\_),n)}$$

$$\frac{\mathsf{safe}_H(r_1,n_1) \quad \mathsf{safe}_H(r_2,n_2) \quad n_1+n_2 \le n}{\mathsf{safe}_H(\texttt{switchMap}(r_1,\_,r_2,\_),n)}$$

$$\frac{\mathsf{safe}_H(r,1)}{\mathsf{safe}_H(\texttt{sub}(r,\_),0)} \qquad \frac{\mathsf{safe}_H(H(r),n)}{\mathsf{safe}_H(r,n)}$$

$$\frac{\forall r.\ H(r)=s=\texttt{sub}(\_,\_) \quad \mathsf{safe}_H(s,0)}{\mathsf{safe}(H)}$$

Figure 12. The safety rules that check the number of event sources guarded by a catch observable or root subscriber.

prevent this is to ensure that all potential sources of errors are protected by a catch observable so that the source of an error can be uniquely identified and the program can possibly recover from the error. To this end, we can check the subscription graph to ensure that each `catchError` (or the root subscriber) can trace to at most one event source without going through another `catchError`.

Figure 12 shows the safety rules to check whether in a heap H, each event source (i.e. `fromEvent`) is uniquely guarded by a `catchError` or root subscriber. The predicate $\mathsf{safe}_H(s,n)$ says that in the subscription s, there are at most n event sources that are not guarded by a `catchError` or `sub`. Given e, if $\emptyset, e \to^* H, v$, we can check the safety of H by checking the predicate $\mathsf{safe}_H(s,0)$ for each $s = \texttt{sub}(x,f)$ in H, where $\to^*$ is the transitive closure of $\to$.

Since the subscription graph in the heap changes during computation, we also need to recheck H when new subscriptions are added. Among the RxJS operators that we considered, only `concat`, `catchError`, and `switchMap` operators will dynamically add subscriptions to the heap. Note that since operators like `map` and `every` can execute arbitrary expression, it can add subscriptions to the heap as well though this is not how they are typically used. Thus, we can check the safety of a RxJS program e as follows:

- If $\emptyset, e \to^* H, v$, then $\mathsf{safe}(H)$ (as defined in Figure 12).
- If $H, Q \to H', Q'$ by Rule $R_{Concat2}$, R_{Catch}, or $R_{Switch3}$, then $\mathsf{safe}(H')$.

For the updated heap, we can just check the new subscriptions instead of the entire heap for better efficiency.

5 Implementation

To provide a testing ground for the proposed semantics, we implemented a subset of RxJS operators using a thread-like abstractions called `AsyncM` [28], which allows asynchronous computation be implemented like a cancellable thread. This implementation includes 100 RxJS operators with about 2500 lines of code (https://github.com/tianzhao/rxjs).

`AsyncM` allows us to chain asynchronous computation just like JavaScript promises except that `AsyncM` can be interrupted via an associated progress object. This abstraction provides a convenient way to implement observables like `fromEvent`, which waits for external events in a loop.

Unlike the operational semantics, the implementation does not need to maintain an explicit queue to hold external events, since `AsyncM` uses promises to handle asynchronous events. Also unlike the semantic, which uses heap variables to maintain bidirectional relation between an observable and its sources, in the implementation, an observable uses variables to access its sources but uses an emitter object to send events to its subscribers.

Subscription. When an observable is subscribed, two threads are launched, which share an emitter and a progress object that are stored in the returned subscription object. This structure is shown in Figure 13.

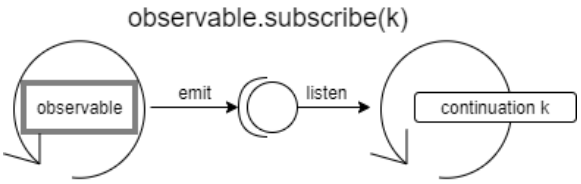

Figure 13. The subscription of an observable, where the circles represent two threads. The left thread runs the observable to emit events and the right thread listens on the emitter and runs the continuation k.

The class for observable has a subscribe method that runs its argument k for each event value until either the end event is emitted or an error has percolated to the top, which cancels the subscription.

5.1 Composite Observable

The composition operators are derived from the methods of observable class. In this section, we explain a few of the operators.

map. The call `o.map(f)` applies a synchronous function f to each event value from `o` except the end event. Any exception in f is caught and emitted as an error event. If the input event is an error, then it is re-emitted by default. This implementation is illustrated in Figure 14.

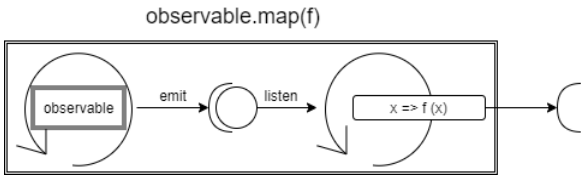

Figure 14. Map the function f to an observable.

switchMap. The call `o.switchMap(f)` applies f to each event of the outer observable `o`, which returns an inner observable. Each time the outer observable emits an event, the current inner observable (if exists) is unsubscribed and a new inner observable is subscribed. This relation is illustrated in Figure 15.

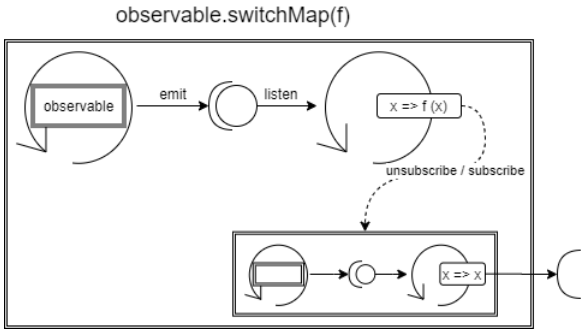

Figure 15. Switch-map the function f to an observable.

The implementation of `switchMap` is similar to that of `map` in that any exception raised in f is caught and emitted as an error event. The subscription to inner observable is referenced as a child of the subscription to the `switchMap`. An error event from the outer or the inner observable is re-emitted by default.

One tricky thing in `switchMap` is to determine when the end is. In RxJS, the end event does not have a value. For example, `of(1,2,3).filter(x=>x<3)` will emit 1 and 2 and then end. Thus, we cannot determine the end of `switchMap` by observing the end of the last inner observable since we do not know whether an inner observable is the last one until the outer observable emits its end event.

The implementation of `switchMap` uses a flag to mark the end of the outer or the inner observable. When the outer observable emits end event, `switchMap` will emit the end event if the flag is set (which indicates that the current inner observable has ended). Otherwise, it will set the flag and wait for the inner observable (which we now know is the last one) to end.

concatAll. A few of the RxJS operators are buffered such as `concatAll`, which concatenates a stream of observables as a single observable. Each inner observable must end before the next one starts. To prevent the loss of the outer observable events, a channel is used. A channel will block its read method if it is empty and block its write method if it is full (if a bound is set). The channel class is also implemented with `AsyncM` so that threads blocked on it can be cancelled.

The `concatAll` operator writes the inner observable emitted from the outer observable into the channel buffer and subscribes to each inner observable read from the channel. The `concatAll` observable ends when an end event is read from the channel.

share. The RxJS observables are not shared by default so that each subscription starts a new instance of an observable. RxJS allows an observable be shared through its `share` operator, which returns a subject that starts running when it is first subscribed and emits events shared by each subsequent subscription. The subject uses reference counting to determine when it should end. That is, when the number of subscriptions drops to zero, the subject terminates.

The subject class derives from the observable class and overrides its internal subscribe method so that multiple subscriptions will listen on events from the same emitter. This design is illustrated in Figure 16.

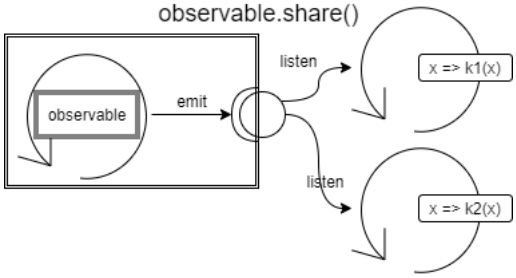

Figure 16. Share an observable as a subject, where multiple subscriptions listen on events from the same emitter.

catchError. An observable may emit error events originated from a primitive or composite observable due to causes such as a rejected promise or an exception in a map function. By default, if an error event is not handled, it will propagate outwards until it reaches the top-level subscriber, when it will cause all subscriptions be cancelled. An error event can be handled with the `catchError` operator, which cancels the source observable and replaces it with a new one.

The `catchError` operator subscribes to its source observable with an error handler that will stop the current subscription and replace it with the subscription to a new observable. The new observable is returned from the argument function f given the error value. However, if the call to f also fails with an exception, then `catchError` emits an error event of its own instead.

Break point. As explained in [1], the difficulties with debugging RxJS program include the inability to set break points for expressions such as `take(n)` and that the stack trace at a break point does not correspond to the dataflow

states of the RxJS program. Our design does provide the ability to inspect the dataflow graphs at break points through the subscription objects. However, we have not implemented a way to add break points to expressions like `take(n)` since it is only used to compose the observable and the subsequent event-handling does not go through the call to `take(n)`.

Stack trace. We have implemented a strategy to capture the subscription graph and the different stack traces when an error occurs. To capture the stack trace, an error handler is used at places that would execute client code such as `fmap` and `switchMap`. The error handler will catch any error thrown from the client code and since the handler is at the topmost part of the stack running in the reactive code, we can easily separate the two types of stack by removing the portion of the stack that share with the handler's stack.

We examine the stack trace by relying on a non-standard feature of all major Web browsers and JavaScript runtime, so it is only possible if the exception value is an instance of the built-in `Error` object. To trace the state of the reactive code, we only need to capture the portion of the subscription graph prior to the error site. This is done by following through the edge to the subscription sources.

Runtime Safety Invariant. The safety invariant is implemented by following the rules in Figure 12. Starting from a root subscriber, we examine the subscription type and make the appropriate assertion by recursively traversing to the source and child subscriptions, checking whether each event source is covered by at least one `catchError` operator. To enable rechecking of the subscription graph when a new subscription is added, each subscription that we came across during the first traversal is assigned to a location accessible to its source/child subscriptions. When a new subscription is added, the subscription assigned previously can be used as a starting point for rechecking the graph.

6 Related Work

RxJS. The reference implementation[1] of RxJS uses Observer Pattern, where an observable passes events to its concrete observers through a generic observer interface. A composition operator acts like a bridge between a source observable and a destination observer where the composition logic is implemented as a decorator of the destination observer. Because a source observable is connected to its intermediate and final observers through references, each event is passed through specific method calls, which includes *next*, *complete*, and *error* events. To check the events between an observable and its intermediate/final observers, a user has to locate and monitor the correct method calls in many classes.

In our design, an observable emits events to its intermediate/final observers through emitters and all types of events pass through the emitter where users can examine a finite history of past events, which include event types, values, and (optional) timing information.

In RxJS, a subscription object unsubscribes by mutating object states. In our design, observables are implemented using `AsyncM` that runs like cancellable threads. In a composite observable, the subscriptions to the inner/source observables are child threads. `AsyncM` supports hierarchical cancellation so that the cancellation of a thread also cancels its child threads. This simplifies the implementation since to unsubscribe an observable, all it takes is to cancel the subscription thread.

Debugging tools for RxJS. This work is motivated by providing helpful debugging information for reactive applications. Due to the complexity of RxJS, users often rely on logging to find errors, which may be helped by logging tools like rxjs-spy[2], which adds tag operators to RxJS so that a trace log can be monitored, paused, and replayed through console. While logging tool reduces debugging workload, trace logs can be difficult to interpret by visual inspection when numerous events are emitted. In comparison, our formal semantics shows that the subscription graph can be inputs to test functions for automated verification and error detection. Visualization tools such as RxFiddle [3], and RxViz[3], and rxjs-playground[4] help visualize the dataflow graph and timing of event emissions through (animated) marble diagrams. These tools are useful for understanding the semantics of RxJS programs but like trace logs, the resulting diagrams have to be manually inspected.

Concurrency. `AsyncM` leverages promises [10] to provide a thread-like concurrency model with cooperative cancellation. While promises do not have builtin methods for cancellation, `AsyncM` enhances the promise constructs with a more consistent way to terminate unused computation.

`AsyncM` is a form of concurrency monad, which is used by Claessen [6] for supporting a simple form of concurrency in Haskell and by Li and Zidancwic [15] in their design for scalable network services. The cancellation mechanism of `AsyncM` is similar to the asynchronous exception of Concurrent Haskell [18], which allows a thread to terminate another thread by throwing an asynchronous exception. However, since JavaScript is not preemptive, the interrupt exceptions in our design are only received at the locations where the threads are blocked or polling the thread status.

The progress object in our design is similar to the cancellation token of *F#* [23, 25] and .Net[5]. The difference is that our design integrates cancellation into a thread model, where a progress (i.e. thread ID) is both the cancellation source and

[1] http://reactivex.io/rxjs

[2] https://github.com/cartant/rxjs-spy
[3] https://rxviz.com/
[4] https://github.com/hediet/rxjs-playground
[5] https://docs.microsoft.com/en-us/dotnet/standard/threading/cancellation-in-managed-threads

token. The hierarchical structure of threads allows a child thread to react to cancellation requests to its parent threads without explicitly linking cancellation tokens.

FRP. Functional reactive programming (FRP) [11, 12] is a framework for modeling continuously changing behaviors that react to discrete events. Classic FRP is pull-based, which detects events by polling in discrete time steps with the implication that the event latency depends on the step size and the behaviors are checked for possible switching every time step. Push-based FRP such as FrTime [7], Flapjax [19], Scala React [17], ReactiveX [20], and Elm [9] provide timely responses to events and avoid re-computation when events do not occur. They wait on event occurrences and only run when an event occurs.

Many research efforts have been devoted to fix problems such as space-time leaks and event glitches using methods such as global dispatcher [19], blocking IO [26], static analysis [14], type-based restrictions [21], mutable memory [24], arrow-based abstractions [5, 8, 16], and a combination of arrow and monad [4, 22]. In practice, however, most of the research designs have not seen wide adoptions like ReactiveX and its JavaScript version RxJS. Despite its flaws, such as the potential of glitches and space leaks, users find its wide range of features appealing.

7 Conclusion

In this paper, we have presented a formal semantics for a selected set of operators in RxJS library. The semantics clarifies the representation of stack trace for RxJS programs, identifies potentially unexpected states of observables, and provides rules for checking sufficient error handling. We provided an implementation of RxJS based on the semantics, which uses an abstraction of cancellable threads to implement observables and their subscriptions. The subscription graph and the events in each graph node are available for debugging purpose.

Since JavaScript is single threaded, there are no simultaneous events. All external events occur in a sequence while internal events are processed when they are generated and before the external events. Thus, our semantics is deterministic, where the same sequence of events to a subscription will trigger the same sequence of reduction rules and result in the same reactive behavior. However, it is unclear whether RxJS is entirely deterministic but using our implementation, programmers can expect predictable outcome.

There are a number of complications in RxJS that are not considered in the semantics. For example, RxJS supports multiple forms of scheduling, which by default is synchronous for internal events. A queue scheduler may be used to process interval events in the order in which they are emitted. RxJS also supports ASAP scheduler that uses the queue for promises and the async scheduler that schedules internal events using event loop or animation frame. Our current semantics corresponds to the default scheduler though it can be easily modified to model the async scheduler. However, separate queues are needed for other type of schedulers. Also, operators like `concatAll` use buffer to hold events until they are used but this may lead to lost events if the buffer is finite or out-of-memory exception if the buffer is unbounded. Shared observables such as a subject can start, stop, or fire events as the side effects of some other observables. These features should be considered to provide accurate modeling of RxJS programs.

References

[1] Manuel Alabor and Markus Stolze. 2020. Debugging of RxJS-Based Applications. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems* (Virtual, USA) *(REBLS 2020)*. Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/3427763.3428313

[2] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

[3] Herman Banken, Erik Meijer, and Georgios Gousios. 2018. Debugging Data Flows in Reactive Programs. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 752–763. https://doi.org/10.1145/3180155.3180156

[4] Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with Type-Level Clocks. *SIGPLAN Not.* 53, 7 (Sept. 2018), 145–157. https://doi.org/10.1145/3299711.3242757

[5] Guerric Chupin and Henrik Nilsson. 2019. Functional Reactive Programming, restated. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*. 1–14.

[6] Koen Claessen. 1999. A Poor Man's Concurrency Monad. *J. Funct. Program.* 9, 3 (May 1999), 313–323. https://doi.org/10.1017/S0956796899003342

[7] Gregory H Cooper and Shriram Krishnamurthi. 2006. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*. Springer, 294–308.

[8] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. 7–18.

[9] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 411–422.

[10] ECMA International. 2015. *ECMA-262: ECMAScript 2015 Language Specification* (6th ed.). Standard. ECMA International. http://www.ecma-international.org/ecma-262/6.0/

[11] Conal Elliott. 2009. Push-pull functional reactive programming. In *Haskell Symposium*. http://conal.net/papers/push-pull-frp

[12] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. 263–273.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA.

[14] Neelakantan R. Krishnaswami. 2013. Higher-Order Functional Reactive Programming without Spacetime Leaks. *SIGPLAN Not.* 48, 9 (Sept. 2013), 221–232. https://doi.org/10.1145/2544174.2500588

[15] Peng Li and Steve Zdancewic. 2007. Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-Level Concurrency Primitives. *SIGPLAN Not.* 42, 6 (June

2007), 189–199. https://doi.org/10.1145/1273442.1250756

[16] Hai Liu and Paul Hudak. 2007. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science* 193 (2007), 29–45.

[17] Ingo Maier and Martin Odersky. 2012. *Deprecating the observer pattern with Scala. react.* Technical Report.

[18] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. 2001. Asynchronous Exceptions in Haskell. *SIGPLAN Not.* 36, 5 (May 2001), 274–285. https://doi.org/10.1145/381694.378858

[19] 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*. 1–20.

[20] Microsoft. 2020. Reactive Extensions. http://reactivex.io/. Accessed: 2020-07-02.

[21] Gergely Patai. 2011. Efficient and Compositional Higher-Order Streams. In *Functional and Constraint Logic Programming*, Julio Mariño (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 137–154.

[22] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional reactive programming, refactored. *ACM SIGPLAN Notices* 51, 12 (2016), 33–44.

[23] Tomas Petricek and Don Syme. 2014. The F# Computation Expression Zoo. In *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages - Volume 8324* (San Diego, CA, USA) *(PADL 2014)*. Springer-Verlag, Berlin, Heidelberg, 33–48. https://doi.org/10.1007/978-3-319-04132-2_3

[24] Atze van der Ploeg and Koen Claessen. 2015. Practical Principled FRP: Forget the Past, Change the Future, FRPNow! *SIGPLAN Not.* 50, 9 (Aug. 2015), 302–314. https://doi.org/10.1145/2858949.2784752

[25] Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The F# Asynchronous Programming Model. In *Practical Aspects of Declarative Languages*, Ricardo Rocha and John Launchbury (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–189.

[26] Atze van der Ploeg. 2013. Monadic Functional Reactive Programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell* (Boston, Massachusetts, USA) *(Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/2503778.2503783

[27] Tian Zhao, Adam Berger, and Yonglun Li. 2020. Asynchronous Monad for Reactive IoT Programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems* (Virtual, USA) *(REBLS 2020)*. Association for Computing Machinery, New York, NY, USA, 25–37. https://doi.org/10.1145/3427763.3428314

[28] Tian Zhao and Yonglun Li. 2021. A Concurrency Model for JavaScript with Cooperative Cancellation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering* (Chicago, IL, USA) *(SLE 2021)*. Association for Computing Machinery, New York, NY, USA, 55–67. https://doi.org/10.1145/3486608.3486911

Received 2022-09-13; accepted 2022-10-10