Concurrency Control of JavaScript with Arrows

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

Adam Berger
bergerab@uwm.edu
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin, USA

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

Abstract

Concurrency control is difficult in JavaScript programs, where event race due to asynchronous computation is a major source of errors. While methods such as promises, cancellation tokens, and reactive programming offer their own advantages in addressing this problem, none offer a complete solution.

In this work, we present an integrated solution for concurrency control of JavaScript using a library, arrowjs, which is based on the abstraction of arrows. Arrowjs uses continuation passing style to chain callbacks and it implicitly generates progress objects to manage concurrency. Arrowjs can implement a form of push-based reactive programming, where event streams are arrow loops communicating through shared memory. Arrowjs thus provides interoperability between thread-like callback chains and event streams with a uniform concurrency control mechanism.

CCS Concepts • **Software and its engineering → Domain specific languages**; *Concurrent programming structures.*

Keywords JavaScript, Concurrency, Arrows, Reactive

ACM Reference Format:
Tian Zhao, Adam Berger, and Yonglun Li. 2019. Concurrency Control of JavaScript with Arrows. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '19), October 21, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358503.3361273

1 Introduction

Concurrency control in asynchronous programs is difficult. Event races are common types of concurrency problems in JavaScript programs where multiple events arrive in an unexpected order or rate resulting in harmful effects [8]. Program analysis is inadequate to debug poorly written programs. Research indicates that even well-tested JavaScript applications often do not adequately cover asynchronous callbacks [5]. Despite the advances in program analysis, static detection of event races may require analysis of the entire JavaScript language [18]. Static analysis of race conditions in JavaScript is fundamentally limited by the complexity of the language itself [13], where the specifics of the happens-before relationship for event handlers often cannot be determined statically, as it may depend on external events [18].

The programming community has adopted promises (and async/await) to provide better concurrency control. With promises, asynchronous tasks can be implemented as thread-like logic without deeply-nested callbacks. However, it does not provide adequate constructs to prevent event races [12]. Separate mechanism such as cancellation token [20] may be used. However, applications with event streams are more naturally implemented with reactive libraries such as rxjs[1], which uses event-based switching for concurrency control. For some applications that include both thread-like logic and event streams, there is a conflict of choice for concurrency control mechanisms.

For example, suppose we want to retrieve the same data from two files, A and B, available from two servers with different formats. To save time, we download and process the files in parallel and use the result that returns first. Suppose that each file is concatenated from parts downloaded from a list of URLs. Also, file A needs to be sorted after download while file B needs to be filtered. Thus, the implementation consists of two parallel tasks. For file A, the task is download, sort, and save. For file B, the task is download, filter, and save. Since only one copy of the data is needed, we should race the two tasks. Using promises, the code might look like

```
Promise.race([download(fileA).then(x=>sort(x))
              download(fileB).then(x=>filter(x))
              ]).then(x=>save(x));
```

(where the `download` function returns a promise that resolves when the file parts are downloaded and concatenated). Unfortunately, this code does not allow the completion event of either download to cancel the other task.

This problem can be solved using two cancellation tokens [20]: T1 and T2. For example, file A task calls cancel on the source of T2 after it completes downloading to notify file B task to cancel. File A task also checks the cancellation

REBLS '19, October 21, 2019, Athens, Greece

ACM ISBN 978-1-4503-6986-2/19/10…$15.00
https://doi.org/10.1145/3358503.3361273

[1] https://github.com/ReactiveX/rxjs

status of T1 when it waits for asynchronous events. Similarly, file B tasks use source of T1 to cancel file A task while monitoring T2 for cancellation notice.

The problem can also be solved with reactive programming. For example, the downloading of a file can be a stream that stops when the completion event of the other file download is received. However, this solution needs to implement the two tasks in a way so that the completion event of one file download can cause cancellation of the other stream. This is unfortunate since the program is written to accommodate cancellation semantics rather than in a style that is more natural to the application itself. While event streams are convenient for file download, each task as a whole is conceptually closer to a thread that should choose its point of progress to trigger the cancellation of the other task.

In this work, we show that a uniform concurrency-control mechanism can be applied to both streaming and thread-like JavaScript code by encoding a push-based reactive interface using arrowjs library [6]. Arrowjs was built upon the work of arrowlets [10] by adding error handling, more flexible concurrency control (noemit), recursive arrow, and type-checking. Arrowjs programs compose asynchronous operations using arrows abstractions [9], which is a generalization of monads [17]. Arrowjs chains asynchronous callbacks by passing the result of the computation to a result continuation and passing any exception to the error continuation. The arrows are also run with implicitly generated progress objects that allow the completion event of an asynchronous arrow to cause cancellation of other racing arrows. With the addition of reactive interface, programmers can choose to use the reactive API for event streams while composing them as arrows for sequential or parallel execution with the same concurrency control mechanism.

Our main contribution is the encoding of a push-based reactive interface using the arrows abstraction with builtin cancellation semantics. We will show that push-based reactive programming can be implemented by modeling each event stream as an arrow loop that sends events to a memory location that may be observed by another event stream. The completion of an event stream is simply the completion of its arrow. Since an event stream is an arrow, it can be part of another arrow that implements a thread-like task that races against other tasks. Like Yampa [2], which uses arrows to transform signals, we use arrows to transform event streams. However, Yampa is demand-driven where events are routed through a loop while our approach is data-driven, where events are propagated through shared memory. Our treatment of events is similar to how reactive values of push-pull FRP [4] are sampled, where a stream processes an event from its sources and then waits asynchronously for the next event to arrive. Unlike other push-based, call-by-value FRP such as Flapjax [16] and Scala.react [14], our events self-propagate, which is similar to those of rxjs though our cancellation semantics is based on progress of arrows.

2 Asynchronous Arrows

This work extends arrowjs [6] with a reactive API so that logic with streaming events can be implemented more naturally. The syntax of arrowjs is shown in Figure 1, where klift for lifting asynchronous function and choice for branching are new additions.

a	::=	f.lift()	lift sync function f
	|	f.klift()	lift async function f
	|	a_1.seq(a_2)	run a_1 and then a_2
	|	a_1.all(a_2)	run both a_1 and a_2
	|	a_1.any(a_2)	race a_1 and a_2
	|	f.choice(a_1, a_2)	branch to a_1 or a_2 using f
	|	a_1.catch(a_2)	catch exception in a_1 with a_2
	|	a.noemit()	make progress only after a
	|	fix($\alpha \Rightarrow a$)	recursive arrow
f	::=	$x \Rightarrow e$	synchronous function
	|	$(x, k) \Rightarrow e$	e passes async result to k
	|	$(x, k_1, k_2) \Rightarrow e$	e calls k_1 or k_2 depending on x
e	::=	...	
	|	a.call(x, p, k, h)	run arrow a with input x, progress object p, result continuation k, and error handler h
	|	a.run()	a.call(null, p, $x \Rightarrow x$, $x \Rightarrow x$)

Figure 1. Syntax of Arrows

Arrowjs programs lift synchronous and asynchronous functions as arrows and composes them to run sequentially, in parallel, or recursively. Each arrow is translated to an object with a call method that takes 4 parameters: an input, a progress object, a result continuation, and an error handler.

A synchronous function f: `a -> b` can be lifted to an arrow of the type `a ~> b` using `lift`: `('a -> 'b) -> ('a ~> 'b)`, where `~>` is arrow type constructor and `'a` and `'b` are type variables. f.lift() wraps the function f in an object with a call method that takes inputs: x, p, k, and h, where x: `a` is the input, p is a progress object, k: `b->_` (where `_` is top type) is a continuation that takes the result of f, and h is a handler that is called with any exception thrown in f.

An asynchronous function f: `(a, b->_) -> _` is lifted using `klift`: `(('a, 'b->_) -> _) -> ('a ~> 'b)`, where f takes an input `x: a` and a callback `k: b->_` to receive the asynchronous response. When a klifted arrow completes, it first checks the progress object to see whether it is cancelled before passing the response to the result continuation and then signals its progress with a call to the progress object so that other arrows racing with this arrow may be cancelled.

For example, in Listing 1, `Arrow.on(elem, evt)` is an arrow that returns a DOM event and `Arrow.delay(n)` is an arrow that returns its input after n milliseconds of delay. A klifted function can return a clean-up function that is

called after k is invoked or when the arrow is cancelled. In this example, the clean-up function removes the event listener. A lifted or composed arrow runs at most once. For example, `Arrow.on('#canvas', 'mousedown')` returns a mouse down event on a canvas at most once (even if the clean-up function is not provided).

```
Arrow.on = (elem, evt) =>
    ((_, k) => {
      /* @arrow :: _ ~> Event */
      $(elem).on(evt, k);
      return _ => $(elem).off(evt, k);
    }).klift();

Arrow.delay = n =>
    ((x, k) => {
      /* @arrow :: 'a ~> 'a */
      setTimeout(_ => k(x), n);
    }).klift();
```

Listing 1. event & delay arrow ($ is an alias for jQuery)

seq: `('a~>'b) -> ('b~>'c) -> ('a~>'c)`[2] is used in a_1.seq(a_2), which runs a_1 and then passes its result to a_2.

all: `('a~>'b) -> ('c~>'d) -> (('a,'c) ~> ('b,'d))` is used in a_1.all(a_2), which runs a_1 and a_2 in parallel and returns their results in a pair.

any: `('a~>'b) -> ('a~>'b) -> ('a~>'b)` is used in a_1.any(a_2), which races a_1 and a_2 and returns the result of one that makes progress first and cancels the other one. The any combinator is implemented by passing a_1 and a_2 a pair of progress objects that observe each other's progress so that if one makes progress then the other one is cancelled.

An asynchronous arrow makes progress when it completes. A composite arrow makes progress when one of its components makes progress. We can suppress the inner progress events of an arrow a with noemit, where a.noemit() only makes progress when a completes. For example, we can define the `until` and `race` methods in Listing 2, where a_1.until(a_2) cancels a_1 if a_2 makes progress and a_1.race(a_2) delays cancellation until a_1 or a_2 ends.

```
Arrow.prototype.until = function(a) {
   return this.noemit().any(a);
};
Arrow.prototype.race = function(a) {
   return this.noemit().any(a.noemit())
};
```

Listing 2. until & race combinator

Listing 3 defines combinators for routing data between arrows such as `carry: ('a ~> 'b) -> ('a ~> ('a, 'b))` that returns the input and output of an arrow and `fanout: ('a~>'b) -> ('a~>'c) -> ('a~>('b,'c))` which is like `all` except both arrows take the same input. Note that `[x,x]` is given a tuple type and `Arrow.id()` is lifted from `x => x`.

```
const split = (x => {
    /* @arrow :: 'a ~> ('a, 'a) */
    return [x, x];
}).lift();

Arrow.prototype.carry = function() {
    return split.seq(Arrow.id().all(this));
}
Arrow.prototype.fanout = function(that) {
    return split.seq(this.all(that))
}
```

Listing 3. carry & fanout combinator

A function f: `('a, 'b->_, 'c->_) -> _` can be lifted to choose to run one of two arrows a_1: `'b~>'d` and a_2: `'c~>'d`, where f.choice(a_1, a_2) : `'a~>'d` runs a_1 if $f(x, k_1, k_2)$ calls k_1 and it runs a_2 if the function calls k_2.

As shown in Listing 4, choice can encode `ifThenElse`:

`('a~>Bool) -> ('a~>'b, 'a~>'b) -> ('a~>'b)`

where a_1.ifThenElse(a_2, a_3) runs a_2 with input of a_1 if a_1 returns true and runs a_3 otherwise.

```
Arrow.prototype.ifThenElse =
    function(thenA, elseA) {
        const choiceA = (([x,b], k1, k2) => {
            if(b) k1(x); else k2(x);
        })
        .choice(thenA, elseA);

        return this.carry().seq(choiceA);
    }
```

Listing 4. branch combinator

The arrow a_1.catch(a_2) catches any exception thrown in a_1, passes it to a_2, and returns its result. Otherwise, the result of a_1 will be passed to the next arrow.

fix : `('a~>'b -> 'a~>'b) -> 'a~>'b` is used in fix($\alpha \Rightarrow a$) that defines a recursive arrow where α is the recursive reference in a. For example, using fix, we can define `forever` that runs an arrow forever and `whileTrue` that runs an arrow repeatedly while it returns true in Listing 5.

```
Arrow.prototype.forever = function() {
       return Arrow.fix(alpha => this.seq(alpha))
};
Arrow.prototype.whileTrue = function() {
       return Arrow.fix(alpha =>
           this.ifThenElse(alpha, Arrow.id()));
};
```

Listing 5. forever & whileTrue combinator

An arrow a: `'a~>'b` is executed with a.run(), which runs an arrow by calling it with a null input, a new progress object, and an identity function as the result/error continuation.

2.1 Autocomplete example

To illustrate the utility of arrowjs, we give an example of adding autocomplete feature to a search box. Autocomplete provides hints to users as they type in characters in a search box. If the hints are queried from a remote source, users may type new characters before the previous query returns, in which case, the pending requests must be cancelled. An arrow implementation is shown in Listing 6, which contains arrows of the following types:

```
keystroke : _ ~> Event
Arrow.delay(1000) : 'a ~> 'a
```

[2]Method is annotated with curried type with receiver as the first argument.

```
  query: Event ~> String
  display: String ~> _
  loop : Event ~> 'a

const keystroke = Arrow.on('#textbox', 'input');

const loop = Arrow.delay(1000) // delay 1 second
  .seq(query)           // send query for hints
  .noemit()             // make progress here
  .seq(display)         // display query result
  .seq(keystroke)       // wait for next input
  .any(keystroke) // cancel delay or pending query
  .forever();         // loop back with input event

keystroke       // first input starts the loop
  .seq(loop)
  .run();           // 'run' starts the arrow
```

Listing 6. Autocomplete

The arrow `keystroke` emits events when a user types in the search box. Then, it starts a loop that waits a second before sending a query with the event and continues to wait for another keystroke. If a keystroke occurs before the delay ends, the query is not sent. If the keystroke occurs after the delay but before the query returns, the result is ignored and the whole process restarts. The 1 second delay ensures that the user has to pause typing for 1 second before the query is sent but as soon as the user starts typing again, any pending timeouts or queries are cancelled.

The cancellation semantics is clearer with parenthesis:

```
( ( Arrow.delay(1000).seq(query).noemit() )
  .seq(display).seq(keystroke) )
.any(keystroke)
```

The arrow in the first and second line runs in parallel with `keystroke` in the third line because of the `any` combinator. An asynchronous arrow makes progress when it completes so that `Arrow.delay(1000)` makes progress when it ends. The same is true for `query`. Since we do not want the delay arrow to cancel `keystroke` until the autocomplete query ends, we wrap both the delay arrow and the query arrow with `noemit` so that it makes progress only after query returns.

Note that when using `a.forever()`, the result of `a` is used as input to run `a` again. In the autocomplete example, the result of the loop body is always the event from `keystroke`.

3 Reactive Interface in Arrows

In this section, we explain the encoding of a push-based reactive API using arrows, where the operators to construct and transform streams are shown in Figure 2.

The autocomplete example in Listing 6 is not easy to understand since it uses a recursive arrow to handle reoccurring keystroke events, but this can be represented more naturally as an event stream. For example, Listing 7 implements autocomplete as event stream, where `Stream.fromEvent` creates a stream from keystrokes, `switch` sends a delayed query for each keystroke while canceling the previous query if it has not completed, `map` displays the query result it receives.

s ::=	repeat(a)	emit output of a forever
|	fromEvent(x, e)	DOM events stream
|	interval(n)	emit its input every n ms
|	forEach(l, a)	run a with each element in l
|	s.take(n)	emit n events of s
|	s.takeUntil(a)	stop s with a's progress
|	s.filter(a)	filter events of s with a
|	s.map(a)	map events of s to a
|	s.mapAsync(a)	map s to a asynchronously
|	s.switch(a)	switch to a for each event of s
|	s.switchMap(s')	switch to s' for each event of s
|	s_1.merge(s_2)	merge events of s_1 and s_2
|	s_1.snapshot(s_2)	sample s_1 for each event of s_2
|	s_1.concat(s_2)	emit events of s_1 and then s_2
|	s.reduce(a)	reduce values from events of s

Figure 2. Stream constructors and operators

```
Stream.fromEvent('#textbox', 'input')
.switch(
      Arrow.delay(1000) // delay 1 second
     .seq(query)        // send query
)
.map(display)       // display the result
.arrow().run();     // run the stream arrow
```

Listing 7. Autocomplete as event stream

There are two challenges in encoding a push-based reactive API with arrows. The first challenge is to construct and transform event streams with recursive arrows. A stream is constructed with a recursive arrow since the klifted arrow for receiving events runs only once. However, for an operation such as taking the first n events of a stream s, we cannot simply combine the recursive arrow of s with another arrow to count the number of events received. To solve this problem, we let each stream emit events to an emitter object to be used by subsequent operators. The second challenge is to monitor the completion status of an asynchronous stream, which completes if the source stream ends and all pending asynchronous operations have ended. To solve this problem, we use counter arrows to monitor the pending operations.

3.1 Stream Constructors

A stream: `Stream 'a 'b = (Emitter 'b) -> ('a ~> 'b)` takes an emitter object and returns an arrow that emits events to the emitter.

The emitter class is defined in Listing 8. Each emitter has an arrow `listenA` that registers a listener for events that are emitted by an event stream via `emitA`. The latest event is stored in the field `now` so that an event stream can be sampled as a behavior by the arrow `nowA`. The emission of events and sampling of the latest event are synchronous while listening for new events is asynchronous (which blocks until the event is received).

```
class Emitter {
    listener = [];   // event consumer
    now = undefined; // latest event

    emitA = (x => {  // send event
        /* @arrow :: 'a ~> 'a */
        this.now = x;
        this.listener.forEach(l => l(x));
        return x;
    }).lift();

    listenA = ((_, k) => { // wait for new event
        /* @arrow :: _ ~> 'a */
        this.listener.push(k);
        return _ => {
            this.listener =
            this.listener.filter(l => l!=k);
        }
    }).klift();

    nowA = (_ => {  // get latest event
        /* @arrow :: _ ~> 'a */
        return this.now;
    }).lift();
}
```

Listing 8. Emitter class

Event streams are defined with `Stream` class in Listing 9. Each stream of the type `Stream 'a 'b` has a function `arrowF` of the type `(Emitter 'b) -> ('a ~> 'b)`. The `instance` method makes an emitter `e`, applies `arrowF` to the emitter, and sequences the resulting arrow with the arrow `e.nowA` to return the last event of the stream (if it is finite).

```
class Stream {
  constructor(arrowF) { this.arrowF = arrowF; }

  instance() {
    const e = new Emitter();
    return [this.arrowF(e).seq(e.nowA), e];
  }
  arrow(){ return this.instance()[0]; }
};
```

Listing 9. Stream class

An event stream can be constructed from an arrow. Listing 10 defines `repeat` that emits the output of an arrow forever, which has the type:

```
('a ~> 'b) -> Stream 'a 'b
```

`fromEvent: (String, String) -> Stream _ Event` makes a stream that emits DOM events and the function `interval: Number -> Stream 'a 'a` returns a stream that emits its input on fixed interval.

```
Stream.repeat = function(arrow) {
  const f = emitter =>
      arrow.seq(emitter.emitA).forever();

  return new Stream(f);
}
Stream.fromEvent = (elem, evt) =>
    Stream.repeat(Arrow.on(elem, evt));

Stream.interval = n =>
    Stream.repeat(Arrow.delay(n));
```

Listing 10. Repeat an arrow forever

A finite stream can be created from an array with `forEach`: `(['a], 'a ~> 'b) -> Stream _ 'b` as shown in Listing 11. This stream runs an arrow with each array element as input and emits the arrow output. Each run of the arrow does not start until the previous run ends. The usage of `times` in Listing 11 is that a.times(n) runs the arrow a with input from 0 to $n-1$. `Arrow.delay(0)` is inserted before event emission to relinquish control of the thread for each array element to make sure that the consumer of this stream has a chance to receive the event.

```
Stream.forEach = function(array, arrow) {
    const elementA =
        (x => {
            /* @arrow :: Number ~> 'a */
            return array[x];
        }).lift();

    const f = emitter =>
        elementA     // return an array element
        .seq(arrow) // run 'arrow' with element
        .seq(Arrow.delay(0))// relinquish control
        .seq(emitter.emitA) // emit 'arrow' output
        .times(array.length); // repeat

    return new Stream(f);
}
```

Listing 11. Finite stream by emitting output of 'arrow' with each element of 'array' as input

3.2 Filtering Operators

We can limit the number of events from a stream with the method `take: Stream 'a 'b -> Number -> Stream 'a 'b` in Listing 12, which repeatedly listens for events of 'this' stream and emits to its own emitter n times. Arrowjs simplifies the semantics of finite streams. The `take` stream completes when either its source stream completes or n events have emitted. The expression `thisArrow.race(loop)` precisely implements these semantics, where `thisArrow` emits source events while `loop` ends when n events have emitted. The completion of either arrow completes the stream.

```
Stream.prototype.take = function(n) {
  const [thisArrow, thisEmitter] = this.instance()

  const f = emitter => {
    const loop = thisEmitter.listenA
                 .seq(emitter.emitA)
                 .times(n);

    return thisArrow.race(loop);
  };

  return new Stream(f);
}
```

Listing 12. Take n events from a stream

`takeUntil: Stream 'a 'b -> (_~>'c) -> Stream 'a 'b` defined in Listing 13 emits events from 'this' stream until its argument 'arrow' makes progress. This implementation simply modifies the arrow of 'this' stream with the `until` combinator so that it ends when 'arrow' makes progress.

```
Stream.prototype.takeUntil = function(arrow){
  const f = emitter => this.arrowF(emitter)
                         .until(arrow);

  return new Stream(f);
};
```

Listing 13. Take events from a stream until the argument 'arrow' makes progress

Events may be filtered with an arrow that takes events as input and returns a Boolean. As defined in Listing 14, `filter : Stream 'a 'b -> ('b ~> Bool) -> Stream 'a 'b` is similar to the `take` method except that `filter` uses the branch combinator `ifTrue` to decide whether to emit an event or skip it. The method `ifTrue` is a syntax sugar, where `a1.ifTrue(a2)` translates to `a1.ifThenElse(a2, Arrow.delay(0))`.

```
Stream.prototype.filter = function(arrow) {
  const [thisArrow, thisEmitter] = this.instance()

  const f = emitter => {
    const loop = thisEmitter.listenA
                 .seq(arrow.ifTrue(emitter.emitA))
                 .forever();

    return thisArrow.race(loop);
  }

  return new Stream(f);
};
```

Listing 14. Emit an event from a stream if 'arrow' returns true with the event as input

3.3 Transformation Operators

`map : Stream 'a 'b -> ('b ~> 'c) -> Stream 'a 'c` transforms an event stream by applying an arrow to each input event and emits the result of the arrow. Unlike libraries such as rxjs and Flapjax and to preserve arrow abstraction, we cannot map to a function that returns streams. The definition of `map` in Listing 15 is similar to that of `filter` except that input events are passed to 'arrow', whose output is emitted.

```
Stream.prototype.map = function(arrow) {
  const [thisArrow, thisEmitter] = this.instance()

  const f = emitter => {
    const loop = thisEmitter.listenA
                 .seq(arrow.seq(emitter.emitA))
                 .forever();

    return thisArrow.race(loop);
  };

  return new Stream(f);
};
```

Listing 15. Map events of 'this' stream to the argument 'arrow' and emit its output

This encoding has a problem: if the source stream completes, the mapped stream completes as well. However, if 'arrow' is asynchronous, the last event is still being processed when the stream completes. We should wait for the last event to complete before terminating the `map` stream.

To solve this problem, we define a counter in Listing 16 to remember the number of pending events.

```
const Counter = function () {
  let count = 0;

  const incA = (x => {
    /* @arrow :: 'a ~> 'a */
    count = count + 1;
    return x;
  }).lift();

  const decA = (x => {
    /* @arrow :: 'a ~> 'a */
    count = count - 1;
    return x;
  }).lift();

  const isPositiveA =( _ => {
    /* @arrow :: _ ~> Bool */
    return count > 0;
  }).lift();

  return [incA, decA, isPositiveA];
}
```

Listing 16. Counter for pending events

The revised definition of `map` is in Listing 17, where we run the arrow `incA` and `decA` before and after running the 'arrow'. We check the counter with `positiveA` after the source stream completes. If the counter is positive, it means there is a pending event and we wait for it using `emitter.listenA` before stopping the `map` stream.

Another problem with the `map` operation is that its loop for listening to source events does not return for the next event until 'arrow' completes, which may be asynchronous and take longer than the time period before the next event arrives. In other words, `map` may miss events if 'arrow' is too slow. These semantics may actually be desirable since it means there is back-pressure and missing events is a potential solution.

```
Stream.prototype.map = function(arrow) {
  const [thisArrow, thisEmitter] = this.instance()

  const f = emitter => {
    const [incA, decA, isPositiveA] = Counter();
    const loop = thisEmitter.listenA // input
                 .seq(incA        // inc counter
                      .seq(arrow) // run arrow
                      .seq(decA)  // dec counter
                      .seq(emitter.emitA)) //output
                 .forever();
    return thisArrow         // wait for last event
          .seq(isPositiveA.ifTrue(emitter.listenA))
          .race(loop);
  };

  return new Stream(f);
};
```

Listing 17. Map operator with proper termination

Alternatively, we can map events to an arrow as soon as the event arrives and emit the result of the arrow in the order in which it completes. This method `mapAsync` is shown in Listing 18, which is similar to `map` except that the event loop

uses spawn : `('a~>'b) -> ('a~>'c) -> ('a~>'b)` to run the argument 'arrow' without waiting for its completion. The arrow a_1.spawn(a_2) runs a_1 and a_2 in parallel with the same input and returns the result of a_1 without waiting for a_2 to complete.

Since an instance of 'arrow' is spawn as soon as a source event arrives, multiple events might be pending when the source stream completes. Thus, we need to run

`isPositiveA.whileTrueThen(emitter.listenA)`

after the source stream completes to wait for event emission as long as the counter is positive. Note that whileTrueThan is a syntax sugar and the arrow a_1.whileTrueThen(a_2) ≡ fix($\alpha \Rightarrow a_1$.ifTrue(a_2.seq(α))).

```
Stream.prototype.mapAsync = function(arrow) {
  const [thisArrow, thisEmitter] = this.instance()

  const f = emitter => {
    const [incA, decA, isPositiveA] = Counter();
    const loop = Arrow.fix(alpha =>
            thisEmitter.listenA // listen input
            .seq(alpha.spawn(
                    incA        // inc counter
                    .seq(arrow) // run arrow
                    .seq(decA)  // dec counter
                    .seq(emitter.emitA))));
                                // emit output
    return thisArrow
     .seq(isPositiveA.whileTrueThen(
                        emitter.listenA))
     .race(loop);
  };
  return new Stream(f);
};
```

Listing 18. Map events to an arrow when they arrive

For example, if we map the input events from a textbox to an autocomplete query and display the result as below, then some input events may be dropped if a query takes too long to return.

```
Stream.fromEvent('#textbox', 'input')
      .map(query)
      .map(display)
```

However, if we change map to mapAsync for the query, then all input events are mapped to the queries but the displayed result may be out of order.

```
Stream.fromEvent('#textbox', 'input')
      .mapAsync(query)
      .map(display)
```

To implement autocomplete as streams, we define switch: `Stream 'a 'b -> ('b ~> 'c) -> Stream 'a 'c` in Listing 19. Each time an event is received from the source stream, the 'arrow' that the switch stream is currently running is cancelled and then restarted with the new event.

```
Stream.prototype.switch = function(arrow) {
  const [thisArrow, thisEmitter] = this.instance()

  const f = emitter => {
    const [incA, decA, isPositiveA] = Counter();
    const loop = Arrow.fix(alpha =>
        thisEmitter.listenA.seq( // wait for event
          incA                   // inc counter
          .seq(arrow)            // run arrow
          .seq(decA)             // dec counter
          .seq(emitter.emitA)    // emit result
          .noemit()     // makes progress here
          .seq(alpha)   // wait for new event
          .any(alpha)   // race against new event
    ));

    return thisArrow
         .seq(isPositiveA.ifTrue(emitter.listenA))
         .race(loop);
  };
  return new Stream(f);
};
```

Listing 19. Switch to run 'arrow' for each event

The event processing loop of switch has the below structure.

```
Arrow.fix( alpha =>
   waitForEvent.seq(
      doSomeWork.noemit()
      .seq(alpha)
      .any(alpha)
))
```

This arrow structure allows us to repeat the processing of waitForEvent so that we can doSomeWork and if it completes, it loops back via seq(alpha) or it may be interrupted by the next event via any(alpha).

In Listing 20, we define a switchMap method of the type `Stream 'a 'b -> Stream 'b 'c -> Stream 'a 'c`, which switches to a new instance of the inner stream for each event of the outer stream. The arrow of the inner stream restarts for each outer event. An arrow is defined to 'pump' the inner events to the emitter of this stream. This 'pump' arrow races with the inner stream in `innerArrow.race(pump)`. Unlike the termination of the switch stream, we cannot wait for the processing of the last outer event since the inner stream may continue to run. Thus, we define a helper emitter 'end' and use `end.emitA` to signal the completion of the inner stream. When the outer stream completes, we use `end.listenA` to wait for the last inner stream to complete.

```
Stream.prototype.switchMap = function(inner) {
  const [thisArrow, thisEmitter] = this.instance()
  const [innerArrow,innerEmitter]=inner.instance()

  const f = emitter => {
    const [incA, decA, isPositiveA] = Counter();
    const end = new Emitter(); // helper emitter
    const pump = innerEmitter.listenA
                 .seq(emitter.emitA).forever();

    const loop = Arrow.fix(alpha =>
      thisEmitter.listenA.seq(
        incA
        .seq(innerArrow.race(pump))
        .seq(decA)
        .seq(end.emitA) // inner stream ends
        .seq(alpha)
        .any(alpha)
      )
```

```
    );

    return thisArrow      // wait for inner to end
          .seq(isPositiveA.ifTrue(end.listenA))
          .race(loop);
  };
  return new Stream(f);
};
```

Listing 20. Switch to an 'inner' stream for each event

We can use `switchMap` to implement drag and drop:

```
Stream.fromEvent('#canvas', 'mousedown')
.filter(isOnTarget)
.switchMap(
    Stream.fromEvent('#canvas', 'mousemove')
    .takeUntil(Arrow.on('#canvas', 'mouseup'))
)
.map(moveTarget)
```

In this example, the mouse-down events are filtered with the `isOnTarget` arrow that returns true iff the mouse-down position is within the target. The arrow `moveTarget` changes the position of the target based on the mouse-move event. The mouse-up event terminates the mouse-move stream.

3.4 Combination Operators

We can merge two streams by emitting the events of either stream as soon as they arrive. The implementation of `merge : Stream 'a 'b -> Stream 'a 'b -> Stream 'a 'b` is in Listing 21. The merged stream completes if both of the component streams complete.

```
Stream.prototype.merge = function(that) {
  const [thisArrow, thisEmitter] = this.instance()
  const [thatArrow, thatEmitter] = that.instance()

  const f = emitter => {
    const loop = thisEmitter.listenA
                .any(thatEmitter.listenA)
                .seq(emitter.emitA)
                .forever()

    return thisArrow.fanout(thatArrow).race(loop);
  }
  return new Stream(f);
}
```

Listing 21. Merge two streams by emitting the latest event of either stream

An event stream can be considered a behavior since its latest event is stored in the emitter. In Listing 22, `snapshot: Stream 'a 'b -> Stream 'a 'c -> Stream 'a ('b,'c)` samples a stream when another stream emits an event. The implementation of `snapshot` only needs to respond to events of 'that' stream and when it happens, it checks the current event of 'this' stream using `thisEmitter.nowA` and pairs it with the emitted event of 'that' stream. The `snapshot` stream ends when 'that' stream ends. To implement these semantics, we use `thisArrow.fanout(loop).race(thatArrow)` which only allows `thatArrow` to complete the execution since `loop` never stops.

```
Stream.prototype.snapshot = function(that) {
  const [thisArrow, thisEmitter] = this.instance()
  const [thatArrow, thatEmitter] = that.instance()

  const f = emitter => {
    const loop = thatEmitter.listenA
      .seq(thisEmitter.nowA.fanout(Arrow.id()))
      .seq(emitter.emitA)
      .forever()

    return thisArrow.fanout(loop).race(thatArrow)
  }
  return new Stream(f);
}
```

Listing 22. Emit the latest event of 'this' stream along with the current event of 'that' stream

Lastly, we show how two streams may be concatenated. `concat: Stream 'a 'b -> Stream 'b 'b -> Stream 'a 'b` in Listing 23 emits events from 'this' stream first and after it completes, it emits events of 'that' stream. We race the arrow of each of the two streams with its own loop for event emission, which terminates when the stream ends. After the loop for emitting events of 'this' stream stops, the loop for emitting the events of 'that' stream starts.

```
Stream.prototype.concat = function(that) {
  const [thisArrow, thisEmitter] = this.instance()
  const [thatArrow, thatEmitter] = that.instance()

  const f = emitter => {
    const thisLoop = thisEmitter.listenA
                .seq(emitter.emitA)
                .forever();

    const thatLoop = thatEmitter.listenA
                .seq(emitter.emitA)
                .forever();

    return thisArrow.race(thisLoop)
           .seq(thatArrow.race(thatLoop));
  }
  return new Stream(f);
}
```

Listing 23. Emits events from 'this' stream before events from 'that' stream

4 Combining Streams and Arrows

Event streams defined in our API can be used in arrows with proper concurrency control. For the file download example in the introduction, we can download file parts using a stream and concatenate the parts in Listing 24.

We assume that `fileListA` and `fileListB` are two lists of URLs and `fetch: String ~> [String]` is an asynchronous arrow that takes a URL as input and returns the content downloaded from the URL as a string array. We also assume `sort` and `filter` are arrows that sort and filter string arrays and `save` is an arrow that saves a string array to a file.

```
const concatenate = ((c, x) => {
    /* @arrow (['a], ['a]) ~> ['a] */
    return c.concat(x);
}).lift();
```

```
const fileA = Stream
    .forEach(fileListA, fetch) \\ download A parts
    .reduce(concatenate) \\ concatenate A parts

const fileB = Stream
    .forEach(fileListB, fetch) \\ download B parts
    .reduce(concatenate)  \\ concatenate B parts

// fileA and fileB race against each other
fileA.arrow().seq(sort)
.any(
    fileB.arrow().seq(filter)
)
.seq(save) // winner of the race is saved
```

Listing 24. Race two file downloads & processing

This example uses the `reduce` method in Listing 25 of the type `Stream 'a 'b -> (('b,'b) ~> 'b) -> Stream 'a 'b`, which applies a reducer arrow to each event from 'this' stream and emits the reduced results.

```
Stream.prototype.reduce = function(reducer) {
  const [thisArrow, thisEmitter] = this.instance()

  const f = emitter => {
    const loop =
      thisEmitter.listenA // wait for input event
      .carry()    // pair reduced value with event
      .seq(reducer)        // apply reduction
      .seq(emitter.emitA) // emit reduced value
      .forever();  //loop back with reduced value

    return thisArrow.race(
            thisEmitter.listenA.seq(emitter.emitA)
            .seq(loop)
    ); //1st event is initial value to reduce loop
  };
  return new Stream(f);
};
```

Listing 25. Reduce events using a reducer arrow

For this example, we are only interested in the final reduced array of the stream. When a stream ends, it outputs its last event. To receive the output, we convert a stream to an arrow before sequencing it with another arrow. In `fileA.arrow().seq(sort)` of Listing 24, we sequence the arrow of `fileA` with `sort`. The output of `fileA` (last event – concatenated array of downloaded file parts) is sent to `sort`.

The race between `fileA` and `fileB` works even if `sort` and `filter` are asynchronous. The use of `any` instead of `race` combinator ensures that the completion of `fileA` or `fileB` cancels the other task immediately. The streams can be part of any racing arrows and can still be cancelled. There is no need to define explicit cancellation tokens or wait until either file to complete processing before cancellation the other. There is no need to make one task to explicitly react to the event of the other task.

5 Related work

Arrows is related to monads [17] and idioms (or applicative functors) [15]. In terms of expressive power, classic arrows are weaker than monads but stronger than idioms [11] though classic arrows extended with *arrow application* is equivalent to monads [9]. Arrowjs may support higher-order event streams with arrow applications while retaining the cancellation semantics.

The arrows abstraction in arrowjs is less general than the classic arrows [9]. For example, f_1.klift().seq(f_2.klift()) behaves the same as $((x,k) \Rightarrow f_1(x, y \Rightarrow f_2(y,k)))$.klift() in isolation, but they have different cancellation semantics when racing with others. The former makes progress when f_1 calls its callback while the latter makes progress when the composite function does. To have the same effect, the former arrow has to be wrapped in an noemit.

Asynchronous programming The addition of promises (and its syntactic sugar async/await) in JavaScript has improved the clarity of asynchronous programs. These constructs are also found in the asynchronous programming model of *F#* [20], which uses monadic abstractions and continuation passing style to pass success, exception, and cancellation continuations, and cancellation tokens for each asynchronous operation. Our implementation of arrowjs is similar except that we use arrows abstraction and instead of passing cancellation continuations and tokens, we pass progress objects, which is similar to chained cancellation tokens that allow a cascading effect of cancellations. Cancellation in racing arrows is triggered automatically via progress made by one of the racing arrow, which can be delayed via noemit or even suppressed. This form of concurrency control is perhaps more suitable for JavaScript since it is single-threaded. The way we use emitters to send and receive events has some similarity to how messages are exchanged in the actor model [7], where an actor waiting on messages suspends itself by storing its continuation and resumes from that continuation when a message becomes available. The difference is that the semantics of the emitter is not fixed and the version we presented does not cache events while new ones such as buffered emitter can provide varying behavior.

Functional reactive programming FRP started as a pull-based design [3] and later research introduced push-pull [4] and push-based [16] models for better performance and lower latency. Events and behaviors can even be unified with clocked signal functions [1]. The way that our stream processes an event and then waits asynchronously for the next event is similar to how reactive values [4] are sampled. Our use of arrows is similar in concept to that of Arrowized FRP such as Yampa [2] and Dunai [19], though we only transform event streams (or piecewise-constant signals) instead of continuous behaviors. Also, our interface is push-based, where each stream pushes events through its emitter while Yampa is pull-based, where external events and behaviors are combined into a single input. Monadic synchronized stream processing such as Dunai and Rhine [1] uses effective signal functions to store side effects including time in monads and avoids event sampling. Though appealing, it is probably difficult to take advantage of this approach in JavaScript.

Event-based FRP such as Flapjax [16] and Scala.react [14] propagates events based on the dependency graph of event streams while we propagate events with operators such as map and merge. rxjs provides a reactive API for managing asynchronous event streams called 'observables'. It features combinators for observables, some of which we have also implemented in arrowjs, for implementing common use cases such as merging event streams, or mapping over the values of a stream. Arrowjs improves upon rxjs by allowing for the same combinators for event streams, while offering implicit cancellation semantics.

Cancellation Semantics In JavaScript, tasks that are pending for events can always be stopped by removing or disabling the callbacks waiting for the events. However, promise does not have an cancellation mechanism like the progress objects of arrowjs or cancellation tokens of *F#*. The race function of promises only discards results of losing promises but does not cancel them when they are pending for events. Cancellation tokens [20] allow for cooperative cancellation of tasks and user-defined cancellation checks. It gives users more control in the context of multi-threaded languages. For JavaScript, the implicit cancellation semantics of arrowjs is perhaps more suitable. Cancellation in reactive programming is based on switching on events. For example, the `untilB` operator in Fran (or `switcher` in push-pull FRP [4]) stops a behavior on an event and replaces the current behavior with a new one. The `switch` function in Yampa [2] replaces one signal-function with another one as a response to events. Later systems such as Flapjax [16] and Scala.react [14] use similar mechanisms. This works well in settings where events are external and a behavior does not stop on its own but is less convenient in settings where events may depend on the completion of finite streams.

6 Conclusion

In this paper, we presented an encoding of a reactive API using arrowjs library. The encoding provides proper cancellation semantics for implementing reactive applications as event streams. It also allows streams to be mixed with arrows code so that they can race with other streams without explicitly reacting to their completion events. This is suitable for concurrency control of applications with thread-like tasks.

There are some possible variations to this design. For example, we can define a buffered stream that uses a buffered emitter with event queues, where the `listenA` arrow registers its callback only when the queue is empty. Higher-order streams may be defined with arrow applications while still preserving the cancellation semantics. Though exceptions thrown in event streams can be caught by enclosing arrows with the `catch` combinator, direct error handling for the stream interface may be added.

References

[1] Manuel Bärenz and Ivan Perez. 2018. Rhine: FRP with type-level clocks. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. ACM, 145–157.

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

[3] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming*.

[4] Conal M Elliott. 2009. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. ACM, 25–36.

[5] Amin Milani Fard and Ali Mesbah. 2017. JavaScript: The (Un)Covered Parts. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. 230–240.

[6] Eric Fritz and Tian Zhao. 2017. Typing and Semantics of Asynchronous Arrows in JavaScript. *Science of Computer Programming* 141 (2017), 1–39.

[7] Philipp Haller and Martin Odersky. 2009. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 410, 2-3 (2009), 202–220.

[8] Shin Hong, Yongbae Park, and Moonzoo Kim. 2014. Detecting Concurrency Errors in Client-Side JavaScript Web Applications. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 61–70.

[9] John Hughes. 2000. Generalising Monads to Arrows. *Science of Computer Programming* 37, 1-3 (2000), 67–111.

[10] Yit Phang Khoo, Michael Hicks, Jeffrey S. Foster, and Vibha Sazawal. 2009. Directing JavaScript with Arrows. In *Proceedings of the 5th Symposium on Dynamic Languages (DLS '09)*. ACM, New York, NY, USA, 49–58.

[11] Sam Lindley, Philip Wadler, and Jeremy Yallop. 2011. Idioms Are Oblivious, Arrows Are Meticulous, Monads Are Promiscuous. *Electron. Notes Theor. Comput. Sci.* 229, 5 (March 2011), 97–117.

[12] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A Model for Reasoning About JavaScript Promises. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 86.

[13] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static Analysis of Event-Driven Node.js JavaScript Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 505–519.

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

[15] Conor Mcbride and Ross Paterson. 2008. Applicative Programming with Effects. *J. Funct. Program.* 18, 1 (Jan. 2008), 1–13.

[16] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 1–20.

[17] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (July 1991), 55–92.

[18] Erdal Mutlu, Serdar Tasiran, and Benjamin Livshits. 2015. Detecting JavaScript Races that Matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 381–392.

[19] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2018. Functional reactive programming, refactored. *ACM SIGPLAN Notices* 51, 12 (2018), 33–44.

[20] Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The F# asynchronous programming model. In *International Symposium on Practical Aspects of Declarative Languages*. Springer, 175–189.