A Concurrency Model for JavaScript with Cooperative Cancellation

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

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

Abstract

This paper proposes a concurrency model for JavaScript with thread-like abstractions and cooperative cancellation. JavaScript uses an event-driven model, where an active computation runs until it completes or blocks for an event while concurrent computations wait for other events as callbacks. With the introduction of Promises, the control flow of callbacks can be written in a more direct style. However, the event-based model is still a source of confusion with regard to execution order, race conditions, and termination.

The thread model is a familiar concept to programmers and can help reduce concurrency errors in JavaScript programs. This work is a library-based design, which uses an abstraction based on the reader monad to pass a thread ID through a thread's computation. A thread can be cancelled, paused, and resumed with its thread ID. This design allows hierarchical cancellation where a child thread is cancelled if its parent is cancelled. It also defines synchronization primitives to protect shared states. A formal semantics is included to give a precise definition of the proposed model.

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

Keywords: asynchronous programming, thread, JavaScript

ACM Reference Format:
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 (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3486608.3486911

1 Introduction

JavaScript provides concurrency through its event loop, where a computation either runs or waits for an event as a listener. As JavaScript applications grow in complexity, it is common to have numerous callbacks with complex dependencies, which makes it difficult to identify concurrent computations. The introduction of Promises [5] allowed the control flow of event callbacks be written in a more direct style. A `Promise` object can encapsulate an event callback. Once constructed, the `Promise` object starts immediately and upon completion, it either resolves successfully with a result or rejects with an error value. When combined with `async` and `await` keywords, the `Promise` abstraction allows asynchronous operations be composed with synchronous operations in a sequential program.

It may seem unnecessary to provide a thread-based concurrency model for JavaScript since `Promises` already behave like threads with non-preemptive scheduling and the methods to wait for their completion such as `race()` and `all()`. However, a `Promise` does not provide utilities for thread synchronization and for cancellation. User-defined constructs for such purposes may not have well-defined semantics, which can result in unexpected behavior. Without proper cancellation, a JavaScript program may contain abandoned computations running in the background, producing unexpected side effects. In this paper, we propose a concurrency model implemented as a library with a formal semantics of thread synchronization and cancellation. A thread, once started, runs until it completes, is blocked on an event, is paused, or is cancelled. A paused thread can be resumed or cancelled.

In this paper, we make the following contributions.

- We motivate the need of a JavaScript concurrency model with cancellation semantics in Section 2.
- We propose a library-based design[1] using the reader monad to represent thread-like abstractions in Section 3. The reader monad implicitly passes a thread ID throughout a computation so that it can be cancelled, paused, and resumed.
- We define a primitive like Haskell's `MVar` and show how it supports communication and cancellation in Section 3.5 and how other primitives such as bounded buffer can be defined in Section 5.

[1]https://github.com/tianzhao/thread

- We give an operational semantics of our concurrency model in Section 4.
- We discuss the usability and overhead in Section 6.

2 Thread-like Concurrency

JavaScript concurrency is based on event handling via its event loop. Event sources behave like stateful objects that dispatch events to registered listeners. The desire to have thread-like behavior resulted in libraries such as node-fibers, which is an implementation of coroutine in node-js with non-preemptive scheduling. This work is not to replicate such capabilities, or suggest a solution of parallel computation using web workers, or leverage the idle time of event-loop for synchronous operations. Our focus is on thread-like abstraction for asynchronous computation, where each thread continues to run until it is blocked, paused, or cancelled. Our goal is help reduce concurrency errors in JavaScript by bringing back the familiar concepts of thread, synchronization primitives, and cancellation mechanisms.

Event races are common types of concurrency errors in JavaScript programs where multiple events arrive in an order or at a rate that is not expected by the programming logic, resulting in unexpected effects [11, 19, 21]. For example, event-race errors can be caused by the concurrent access to an external resource (e.g. a web service) if it does not protect against such access. These errors can be difficult to debug since they are reflected in the incorrect states at the external resource instead of at the JavaScript program. Furthermore, research indicates that even well-tested JavaScript applications often do not adequately cover event-dependent or asynchronous callbacks [8], inviting alternative methods to identify issues in such constructs.

While Promises [5] have helped reduce deeply-nested callbacks, its semantics is still complex [14, 17] and the number and breadth of issues reported on platforms like Stack Overflow indicate that users often struggle to understand its proper use. Static methods like Promise Graph [15] were proposed to track when Promises are defined and activated/resolved as a step toward helping developers identify issues, it still does not indicate when pieces may execute in parallel and may cause concurrency errors.

We argue that the thread abstraction has several advantages over Promises.

- The first advantage is conceptual. To run an operation in a separate thread, one must start the thread explicitly. However, a Promise object is concurrent by default. If a Promise object is run for its side effect, one can easily forget to sequence it (e.g. using *await*) without realizing that it may run in a different order.
- Secondly, since threads have a well-defined abstraction boundary, it is easier to recognize concurrent access to shared resources so that synchronization primitives can be used to protect their access.
- Thirdly, Promise objects do not have methods for cancellation. While we can use the race() method (which waits for and returns the first value produced by a collection of Promise objects) to implement a task such as to timeout an asynchronous operation, the operation does not actually stop – only its results are abandoned. Though JavaScript is not preemptive, cancelling a thread at the earliest opportunity can reduce unintended side effects from the abandoned operations.

3 Concurrency with Cancellation

Promises are similar to the continuation monads for concurrency [4, 13], which allow asynchronous callbacks be chained together without deeply nested scopes. One difference is that a Promise object, once constructed, starts immediately while continuation monads are started explicitly.

```
new Promise((resolve, reject) => {
    // call resolve with results
    // or call reject with error
)}
```

A Promise is instantiated with an executor function with two parameters: *resolve* and *reject*. A Promise runs exactly once, which results in a fulfilled state (if *resolve* is called) or a rejected state (if *reject* is called or an exception is thrown). If neither functions are called, then the Promise object remains in a pending state. Promises can be sequenced using *then* method and exceptions can be handled using *catch* method.

```
p.then(x => { /* returns a promise */ })
 .catch(e => { /* handle error */ })
```

Our design is a cancellable concurrency monad with a thread ID. We implement it as a reader monad that wraps a function that takes a thread ID (of the type Progress) and returns a Promise object.

```
Progress -> Promise a
```

We define a JavaScript class AsyncM to represent this concurrency monad.

```
class AsyncM {
  // run :: Progress -> Promise a
  constructor (run) { this.run = run; }

  // pure :: a -> AsyncM a
  static pure = x => new AsyncM (
     p => Promise.resolve(x)
  )
  // fmap :: AsyncM a -> (a -> b) -> AsyncM b
  fmap = f => new AsyncM (
     p => this.run(p).then(f)
  )
  //bind :: AsyncM a -> (a->AsyncM b) -> AsyncM b
  bind = f => new AsyncM (
    p => this.run(p).then(x => f(x).run(p))
  )
}
```

The static method pure converts a constant value to an AsyncM that always return that value. The fmap method applies a function to this AsyncM while the bind method composes this AsyncM with a function that returns an AsyncM. These methods allow AsyncMs to be composed so that a Progress value can be passed implicitly through the AsyncM computation. This style of composition does come with syntactic overhead since it prevents the direct use of async and await keywords for composing Promises.

```
// lift :: ((a -> ()) -> ()) -> AsyncM a
static lift = f => new AsyncM (p =>
  new Promise((resolve, reject) => {
    // cancel by throwing an exception
    let c = _ => reject("interrupted");

    if (!p.cancelled) { // check thread status
      p.addCanceller(c); // add a canceller

      // remove canceller when 'f' completes
      let k = x => {
        p.removeCanceller(c)
        resolve(x);
      }
      f(k) // starts an asynchronous operation
    }
    else c(); // cancel if the thread is dead
  })
)

static timeout = n => AsyncM.lift(k =>
                                setTimeout(k, n))
```

An AsyncM that runs an asynchronous operation is constructed with the lift method, whose parameter f performs an asynchronous operation such as setTimeout. The lift method also enables cancellation, which may happen when this AsyncM is blocked on its call to f. The AsyncM returns a resolved Promise if f completes with a result and it returns a rejected Promise if it is cancelled. For simplicity, the error handling of f is not described, which involves passing f a handler h so that if f raises an error e, then h removes the canceller c and calls the reject function with e.

3.1 Thread ID and Cancellation

The thread IDs are used for cancellation. A thread ID is a Progress object, which forms a tree, where each tree node has a cancellation flag and a set of canceller functions.

```
class Progress {
  constructor(parent) {
    if (parent) {
      this.parent = parent;
      parent.children.push(this);
  }}
  cancelled = false
  children = []
  cancellers = [] }
```

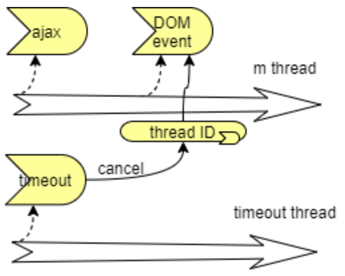

Figure 1. Thread cancellation through timeout.

To start a thread, we simply run an AsyncM with a new Progress object and return it.

```
class AsyncM {
    start = _ => {
        let p = new Progress()
        let f = _ => this.run(p)
        setTimeout(f, 0) // start f asynchronously
        return p
}}
```

When a lifted AsyncM is run with a Progress object p, a canceller function is added to p. If the AsyncM completes, the canceller is removed. If the AsyncM is cancelled before its completion, then the canceller runs, which causes the AsyncM to return a rejected Promise.

For example, the variable m below fetches data using an $ajax$ call, performs some computation, and sends the results for display. We start m with a thread ID t, which is used to cancel m if it is not completed within a second.

```
let m = AsyncM.lift(ajax) // fetch data
        .fmap(compute) // synchronous action
        .bind(display) // visualize data

let t = m.start() // starts m with thread ID t

AsyncM.timeout(1000)
      .fmap(_ => t.cancel())
      .start()   // starts a timeout thread
```

As shown in Figure 1, the timeout thread references the thread ID of m and may use it to cancel m when m is blocked on an event (e.g. when m calls the ajax or display function). However, like other non-preemptive designs, the timeout will not have immediate effects if it occurs while a synchronous operation like compute is running.

Unlike a Promise object, which runs immediately after its composition, the composition of an AsyncM object is separate from its execution, which helps identify the concurrent operations. One can delay the execution of a Promise by defining a function that returns a Promise (such as an async function). However, there is no syntactic difference between calling an async function and calling a regular function. It is easy to forget the difference between calling an async function with or without using await to wait for its completion.

3.2 Asynchronous Exception

When a thread ID t is cancelled, an interrupt exception is sent to the thread running with t. This design is similar to the asynchronous exception of Concurrent Haskell [16], except that our interrupt exception can only be received at some locations. In our case, the exception is received immediately if the thread is blocked, resulting in a rejected `Promise`. Otherwise, the exception is received when the thread makes a blocking call or checks the status of the thread ID. For example, if a thread cancels another thread, the effect is immediate. However, if a thread cancels itself, there may be some delay.

The interrupt exceptions may be received by an `AsyncM` such as `AsyncM.lift(f)`. When this `AsyncM` runs, it first checks whether its `Progress` p is alive. If it is not alive, then it returns a rejected `Promise`. Otherwise, it adds a canceller c to the progress object p so that if p is cancelled, then c will be called to cause an interrupt exception.

The interrupt exception of the previous example can be handled with the *catch* method as shown below.

```
class AsyncM {
  catch = h => new AsyncM(p =>
                           this.run(p).catch(h))
}
let m = AsyncM.lift(ajax) // fetch data
        .fmap(compute) // synchronous action
        .bind(display) // visualize data
        .catch(print)  // print exception
```

3.3 Fork and Hierarchical Cancellation

Other than starting an independent thread, we can also fork a child thread with a `Progress` object that is a child of the current `Progress`. The parent `Progress` has a reference to the child progress so that if the parent is cancelled, so is the child. When the forked thread completes, the reference from a parent `Progress` to its child `Progress` is removed using the `unlink` method to avoid memory leak.

```
class AsyncM {
  fork = _ => new AsyncM (async p => {
      const p1 = new Progress(p)
      // start the thread asynchronously
      // unlink the reference from p to p1
      // after the thread completes
      AsyncM.timeout(0).bind(_ => this).run(p1)
            .finally(_ => p1.unlink());
      return p1;
  })
}
class Progress {
  // remove the parent to child reference
  unlink = _ => {
      let p = this.parent
      if(p) p.children = p.children
                          .filter(c => c != this)
  }
  // call all cancellers recursively
  cancel = _ => {
    this.cancelled = true // set cancel flag
    this.cancellers.forEach(c => c())
    this.children.forEach(c => c.cancel())
    this.cancellers = [] // clear cancellers
  }
}
```

The `cancel` method of the `Progress` class sets the cancellation flag, calls each registered canceller to signal interrupts, and recursively cancels its children.

Using `fork`, we can run the m thread in the last example as a child of the timeout thread so that both threads can be cancelled by a user action such as pressing a "stop" button.

```
// run 'm' as a child of the timer thread
let t = m.fork()
         .bind(t1 => AsyncM.timeout(1000)
                       .fmap(_ => {
                          t1.cancel()
                          console.log("timeout")
                       })
         ).start()

// user cancels 'm' and the timer thread
$("#stop").one('click', _ => t.cancel())
```

The above example has 3 possible outcomes:

1. m completes,
2. m is cancelled by the timer, which prints 'timeout' message, and
3. user stops both m and the timer thread.

If it is inconvenient to use `fork` and `bind`, one can also run threads directly with `Progress` objects as shown below.

```
let t = new Progress()
let t1 = new Progress(t)

m.run(t1) // run m with t1

AsyncM.timeout(1000)
      .fmap(_ => {
           t1.cancel()
           console.log("timeout")
      }).run(t) // run timer with t

$("#stop").one('click', _ => t.cancel())
```

We can also use a `Progress` like a cancellation token. For example, we can start a set of threads by calling their `run` methods with a `Progress` t or the children of t so that any thread in the group can cancel the group through t.

3.4 Pause and Resume Threads

Our threads can be paused and resumed. This is useful in cases such as debugging and the implementation of user interfaces. For example, users can pause threads in browser console to check the states of the program or add controls to user interface to pause and resume animation threads such as real-time data charts.

```
$("#pause").on('click', _ => t.pause())
$("#resume").on('click', _ => t.resume())
```

We can add buttons to the previous example to allow users to pause and resume the timer and *m* threads. Unlike thread cancellation, which throws exceptions to the cancelled threads, thread suspension is based on polling. A thread returning from a blocking call checks whether it is paused and if so, it adds its continuation to the progress object, on which the pause method is called. This means that pausing a thread does not suspend its asynchronous calls but the handlers to the calls.

Like cancellation, thread suspension is hierarchical. If we pause a thread *t*, then *t* and its children are paused. Also, while a paused thread can be cancelled, a cancelled thread cannot be resumed. Pausing a cancelled or completed thread has no effect. For our example, if `t.pause()` is called before the timer and *m* complete, then *m* may be suspended after the `ajax` or `display` call returns while the timer thread will not advance beyond the timeout.

To reduce unintended side effects, a thread can only be resumed by the same `Progress` that the thread is paused with. That is, the threads that are paused together can only be resumed together. For example, if *m* is paused by the call `t.pause()`, then it can only be resumed by the call `t.resume()`. A thread can be paused or resumed by any code with access to its thread ID. Thus, it is possible that a paused thread *t* can remain paused forever, if the thread that will resume *t* is cancelled.

```
static lift = f => new AsyncM (
  p => new Promise(
    (resolve, reject) => {
      let c = _ => reject("interrupted");

      if (!p.cancelled) {
        p.addCanceller(c);

        let k = x => {
          p.removeCanceller(c)

          // suspend the thread if it is paused
          if (! p.isPaused(_ => resolve(x)))
            resolve(x);
        }
        f(k)
      }
      else c();
    }
  )
)
```

The `lift` method above is revised to poll the pause status of a thread. It checks whether its `Progress` *p* is paused after the lifted function f returns and if so, it adds the `resolve` continuation to *p*.

```
class Progress {
  paused = false // pause status flag
  pending = []   // pending thread continuations

  pause = _ => { this.paused = true }

  isPaused = k => { // k: thread continuation
    if (this.paused) {
      this.pending.push(k)
      return true
    }
    else if (this.parent) {
      return this.parent.isPaused(k)
    }
    else {
      return false
    }
  }
  // resume paused threads
  resume = _ => {
    this.paused = false
    if (! this.cancelled)
      this.pending.forEach(k => setTimeout(k, 0))
    this.pending = []
  }
}
```

The `Progress` class is added a pause-status flag and a list of the pending thread continuations. The `pause` method simply sets the status flag to true while the `isPaused` method checks the status of this progress and its ancestors recursively and adds the continuation *k* is to the paused `Progress`. The `resume` method restarts paused threads by calling the pending continuations after a timeout.

3.5 Synchronization Mechanism

In JavaScript, an event race can be caused by the concurrent access to shared resources. To help prevent event races, we include synchronization primitives similar to Haskell's `MVar`, which can be used as locks to protect resources from concurrent access. For example, in a real-life application[2], a bug was caused by an user interface that sends concurrent requests to a remote service without support for concurrent access. If a user sends a new request before the previous request completes, then the new request will cause an internal error in the remote service.

The code below illustrates this problem, where the response to the request is sent to an user callback *cb*.

```
$("#button").on("click", _ => sendRequest(cb))
```

A simple fix is to use a flag to stop the handler from responding to the button click before a request completes.

[2] https://github.com/TryGhost/Ghost/issues/1834

```
let flag = true
$("#button").on("click", _ => {
    if (flag) {
        flag = false;
        sendRequest(x => {
            flag = true
            cb(x)
        })
    }
})
```

However, this fix is not ideal since some clicks would lead to a response while others do not. If we want each button click to trigger a response safely, we can use a synchronization primitive like MVar.

A MVar object m can hold one value and is either empty or full. A thread that puts value in m blocks if m is full. A thread that takes value from m blocks if m is empty. If multiple take (or put) threads are blocked on m, only the first one on queue is allowed to proceed after m is filled (or emptied).

```
let m = new MVar()              // create a lock
$("#button").on("click",  _ =>
         m.put(0)               // obtain the lock
          .bind(_ => AsyncM.lift(sendRequest))
          .bind(x => m.take() // release lock
                      .fmap(_ => cb(x)))
          .start()
)
```

In the code above, m is used as a lock to ensure that sendRequest is called once at a time. If the button is clicked before the previous request completes, the new request will be blocked on m until the previous request releases the lock.

Like threads blocked on events, threads blocked on a MVar can also be cancelled. The put method shown below adds a canceller c to the Progress object p if it is blocked (i.e. the MVar is full) and the canceller is removed when the thread unblocks (i.e. MVar is emptied). The canceller would remove the thread from the list of threads pending on the MVar and throw an exception.

```
class MVar {
  isEmpty = true;
  pending = []; // pending put or take threads

  // put :: MVar a -> a -> AsyncM ()
  put = x => new AsyncM(p => new Promise(
   (resolve, reject) => {
    if (!this.isEmpty) { // block if not empty
      let k = _ => { // 'put' continuation
        p.removeCanceller(c)
        this._put(x)
        setTimeout(resolve, 0) // resume later
      }
      let c = _ => { // removes pending thread
        this.pending =
             this.pending.filter(t => t != k)
        reject("interrupted"); // raise exception
      }
      p.addCanceller(c) // enable cancellation
      this.pending.push(k)
    }
    else { // put 'x' and continue if empty
      this._put(x)
      resolve()
    }
  }))
  // put 'x' in MVar when it is empty
  _put = x => {
    this.isEmpty = false
    this.value = x

    // wake up a pending 'take' thread
    if (this.pending.length > 0)
      this.pending.shift()()
  }

  // the 'take' method is similar
}
```

The take method (details omitted) registers a canceller (identical to that of the put method) with the Progress p if it is blocked (i.e. the MVar is empty). This canceller will be removed when the take thread unblocks.

Like other synchronization mechanisms, our MVar is susceptible to deadlocks. For example, a take thread blocked on an empty MVar m is in a deadlock state if it also holds locks that prevent other threads from putting data in m. However, since JavaScript is not preemptive, it is less likely to enter a deadlock state due to non-determinism than a language with preemptive scheduling.

It may be possible to simulate priority-based scheduling by assigning a priority level to each thread when it is started. MVar could be modified so that it wakes up the pending threads based on their priorities. A thread can then yield to other threads by blocking itself on the modified MVar.

4 Operational Semantics

In this section, we formalize our design by giving an operational semantics in the style of Concurrent Haskell. This semantics includes two sets of rules: asynchronous rules for the reduction of AsyncM, which encodes thread computation and is possibly blocking, and synchronous rules for other non-blocking computation.

In Figure 2, we define terms and values. The symbol V ranges over values such as constants, thread ID, MVars, functions, and AsyncM values.

The symbol A ranges over AsyncM which includes primitives such as $\text{pure}(V)$ that returns value V, $\text{throw}(c)$ that throws an error c, and $\text{lift}(f)$ that runs asynchronous function f and waits for its results. AsyncM also includes combinators: $A.\text{bind}(f)$ that passes the value of A to f, $A.\text{catch}(f)$ that catches the error of A with f, and $A.\text{fork}()$ that runs A in a child thread.

f	$::=$	x => M	Functions
A	$::=$		AsyncM values
	$\mid$	pure(V)	return value
	$\mid$	throw(c)	throw exception
	$\mid$	lift(f)	asynchronous action
	$\mid$	A.bind(f)	monadic bind
	$\mid$	A.catch(f)	handle exception
	$\mid$	A.fork()	fork a child thread
	$\mid$	m.put(V)	put value in MVar
	$\mid$	m.take()	take value from Mvar
V	$::=$		Value
	$\mid$	c	constant
	$\mid$	undef	undefined value
	$\mid$	t	thread ID (progress)
	$\mid$	m	MVar
	$\mid$	f	
	$\mid$	A	
M, N	$::=$		Terms
	$\mid$	V	
	$\mid$	x	variable
	$\mid$	M.start()	start a thread
	$\mid$	M.cancel()	cancel a thread
	$\mid$	M.pause()	pause a thread
	$\mid$	M.resume()	resume a thread
	$\mid$	new `MVar`	allocate `MVar`
	$\mid$	$M\ (N)$	call
	$\mid$	$M\ ?\ N_1 : N_2$	branch
	$\mid$	$\ldots$	
t	$::=$		Thread ID
	$\mid$	p	root progress
	$\mid$	$t \cdot p$	child progress
u	$::=$		Main or thread ID
	$\mid$	ϵ	ID of main
	$\mid$	t	

Figure 2. The syntax of `AsyncM`, values, and terms

The symbols M and N range over terms, which include values, function call, branch, new `MVar`, and the term to start/cancel/pause/resume a thread. We omit other terms such as M.bind(f), which should be reduced to the value A.bind(f) before being reduced as a monad.

The symbol t ranges over thread ID, which is either a root `Progress` p or a child `Progress` $t \cdot p$ with t as the parent. For the main program, we use ϵ to denote its ID.

4.1 Program Transitions

We define our semantics by describing the transitions between program states. A program state (Figure 3) is a parallel composition of threads, `MVar`s, and the cancel/pause/resume actions on threads.

P, Q, R	$::=$		States
	$\mid$	$\llparenthesis M \rrparenthesis_u$	live thread with ID u
	$\mid$	$\llparenthesis M \rrparenthesis^{\bullet}_t$	stuck thread
	$\mid$	$\llparenthesis M \rrparenthesis^{\circ}_t$	ready thread
	$\mid$	$\llparenthesis\rrparenthesis_u$	completed thread
	$\mid$	$\langle\rangle_m$	empty `MVar` named m
	$\mid$	$\langle V \rangle_m$	`MVar` m filled with V
	$\mid$	$\llbracket t \lightning \text{cancel} \rrbracket$	t is cancelled
	$\mid$	$\llbracket t/t_i \lightning \text{pause} \rrbracket$	t_i is paused by t
	$\mid$	$\llbracket t/t_i \lightning \text{resume} \rrbracket$	t_i is resumed by t
	$\mid$	$\nu x.P$	restriction
	$\mid$	$P \mid Q$	parallel composition

Figure 3. The syntax of program states

$$
\begin{array}{rcll}
P \mid Q & \equiv & Q \mid P & \text{(Comm)} \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & \text{(Assoc)} \\
\nu x.\nu y.P & \equiv & \nu y.\nu x.P & \text{(Swap)} \\
(\nu x.P) \mid Q & \equiv & \nu x.(P \mid Q),\ x \notin fn(Q) & \text{(Extrude)} \\
\nu x.P & \equiv & \nu y.P[y/x],\ x \notin fn(P) & \text{(Alpha)}
\end{array}
$$

Figure 4. Structural congruence

A thread is either alive $\llparenthesis M \rrparenthesis_u$, stuck $\llparenthesis M \rrparenthesis^{\bullet}_t$, or ready $\llparenthesis M \rrparenthesis^{\circ}_t$, where the subscript is the thread ID. A live thread is currently running, the stuck threads are waiting for events or blocked on `MVar`s, and a ready thread will run when there is no other live thread. The initial program state is the main thread $\llparenthesis M \rrparenthesis_\epsilon$.

A program state can transition to the next state with or without a label, which is written as: $P \xrightarrow{\alpha} Q$. The label, if present, represents an asynchronous event c received by the JavaScript event loop: $P \xrightarrow{?c} Q$.

The transitions of the program states are supported by the equivalence relation $\equiv$ defined in Figure 4 – identical to the one in Concurrent Haskell [16, 20].

The structural transitions of program states are defined in Figure 5. Since JavaScript is not preemptive, there can be at most one live thread at any time. To model this behavior, we place a restriction in Rule (Par) so that the transition from P to Q can cause transition from $P \mid R$ to $Q \mid R$ only if R does not contain live threads. In other words, while a live thread is running, no other threads can become alive.

$$\mathsf{live}((\!|M|\!)_u) \quad \frac{\mathsf{live}(P)}{\mathsf{live}(\nu x.P)} \quad \frac{\mathsf{live}(P)}{\mathsf{live}(P \mid Q)} \quad \frac{\mathsf{live}(Q)}{\mathsf{live}(P \mid Q)}$$

$$\frac{P \xrightarrow{\alpha} Q \qquad \neg\mathsf{live}(R)}{P \mid R \xrightarrow{\alpha} Q \mid R}\ \text{(Par)} \qquad \frac{P \xrightarrow{\alpha} Q}{\nu x.P \xrightarrow{\alpha} \nu x.Q}\ \text{(Nu)}$$

$$\frac{P \equiv P' \qquad P' \xrightarrow{\alpha} Q' \qquad Q' \equiv Q}{P \xrightarrow{\alpha} Q}\ \text{(Equiv)}$$

Figure 5. Structural transitions

4.2 Transition Rules

This section explains the transition rules for AsyncM values (Figure 6), the rules for terms (Figure 7), and the rules for the actions on threads (Figure 8). The transition rules for AsyncM values describe the thread computation, which takes place within an evaluation context $\mathbb{E}$ defined below.

$$\mathbb{E} \quad ::= \quad [\cdot] \mid \mathbb{E}.\mathsf{bind}(f) \mid \mathbb{E}.\mathsf{catch}(f)$$

Bind and Catch. Rule (Bind) describes how a value is passed to the sequenced function. Rule (Catch) describes how an error is caught by a handler. The two propagate rules describe how values and errors are propagated through catch and bind.

Fork. Rule (Fork) says that a child thread is forked with an ID that is the child of the current ID. The child thread starts in the ready state (denoted by the superscript $\circ$) so that the parent thread can continue to run.

Async. Rules (Stuck-Async) and (Async) describe how the expression $\mathsf{lift}(f)$ runs the asynchronous function f and waits for its result c as an event. $\mathsf{lift}(f)$ first transitions to a stuck state denoted by the superscript $\bullet$. After receiving an event, the stuck thread $(\!|\mathbb{E}[\mathsf{lift}(f)]|\!)^{\bullet}_t$ transitions to a state that returns the event value c.

MVar. Rule (Put-MVar) says the thread $(\!|\mathbb{E}[m.\mathsf{put}(V)]|\!)^{b}_t$ fills an empty MVar $\langle\rangle_m$ with its value V, where the superscript b means that the thread is either alive or stuck. If m has value, then by Rule (Stuck-Put-MVar), $m.\mathsf{put}(V)$ transitions to a stuck state. Rules for $m.\mathsf{take}()$ are similar.

Terms. The transitions of terms (shown in Figure 7) take place within the context $\mathbb{F}$ defined below.

$$\begin{aligned}\mathbb{F} \quad ::= \quad & [\cdot] \mid \mathbb{F}\ (M) \mid f\ (\mathbb{F}) \mid \mathbb{F}\ ?\ M_1 : M_2 \mid \\ & \mathbb{F}.\mathsf{start}() \mid \mathbb{F}.\mathsf{fork}() \mid \\ & \mathsf{pure}(\mathbb{F}) \mid \mathbb{F}.\mathsf{bind}(f) \mid \mathbb{F}.\mathsf{catch}(f) \mid \\ & \mathbb{F}.\mathsf{put}(M) \mid m.\mathsf{put}(\mathbb{F}) \mid \mathbb{F}.\mathsf{take}() \mid \\ & \mathbb{F}.\mathsf{cancel}() \mid \mathbb{F}.\mathsf{pause}() \mid \mathbb{F}.\mathsf{resume}()\end{aligned}$$

The terms include expressions like new MVar, function call, and branch, whose transitions are non-blocking. That is, a live thread will continue to be alive after the transition. Rule (Start) describes how $A.\mathsf{start}()$ starts a new thread with a root progress as its thread ID. The new thread is also in the ready state so that the calling thread can continue to run.

Run and Termination. By Rule (Run), a thread in ready state can transition to a live thread; and by Rule (Par) in Figure 5, this transition is allowed if there is no other live threads. This semantics is implemented by running the thread with a 0 second timeout so that a ready thread is scheduled to run by the JavaScript event loop after other threads complete or become stuck. Figure 7 also includes the rules for thread termination, which states that a forked or started thread terminates if it returns a value or throws an error while the main thread terminates when it reduces to a value. Rule (GC) says that a terminated thread is removed from the program.

Thread Actions. Figure 8 defines the transition rules for terms that cancel, pause, and resume threads. Rule (Cancel) states that the term $t.\mathsf{cancel}()$ reduces an undefined value while producing actions to cancel threads with ID t_i that satisfies the relation $\mathsf{prefix}(t, t_i)$. The actions are denoted by the set $\{[\![t_i \lightning \mathsf{cancel}]\!]\}_{\mathsf{prefix}(t,t_i)}$. The relation $\mathsf{prefix}(t, t_i)$ is defined below, which means that t either equals t_i or is an ancestor of t_i.

$$\mathsf{prefix}(t,t) \qquad \frac{\mathsf{prefix}(t,u)}{\mathsf{prefix}(t, u \cdot p)}$$

In other words, $t.\mathsf{cancel}()$ will cancel any threads running with t or the children of t as thread IDs. By Rule (Cancel-Stuck), the cancel action will cause a stuck thread to throw an exception. This rule takes precedence over Rule (Async) so that a stuck thread, if cancelled, will always terminate. By Rule (Cancel-Async), the cancel action will cause a live thread with an asynchronous operation to throw an exception. The rule is applicable when a thread is cancelling itself.

Rule (Pause) describes how $t.\mathsf{pause}()$ reduces to the undef value in its context and produces a set of the pause actions $\{[\![t/t_i \lightning \mathsf{pause}]\!]\}_{\mathsf{prefix}(t,t_i)}$. By Rule (Pause-Stuck), each of the pause actions can pause a stuck thread when it unblocks, which transitions to another stuck thread $(\!|\mathbb{E}[\mathsf{pure}(c)]|\!)^{\bullet,t}_{t_i}$. The superscript t indicates that this thread can only be resumed by the call $t.\mathsf{resume}()$ according to the rules (Resume) and (Resume-Paused).

5 Additional Constructs

Race and All. We provide a race combinator to conduct a race among a list of threads and an all combinator to run a list of threads concurrently and to wait for their results. The composed threads can be cancelled altogether without any additional logic due to our cancellation mechanism.

If we race a list of threads, the losing threads will also be cancelled. The race method below first allocates a child

$$
\begin{array}{rcll}
\llparenthesis \mathbb{E}[\mathsf{pure}(V).\mathsf{bind}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[f(V)] \rrparenthesis_t & \text{(Bind)} \\
\llparenthesis \mathbb{E}[\mathsf{pure}(V).\mathsf{catch}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[\mathsf{pure}(V)] \rrparenthesis_t & \text{(Propagate-Value)} \\
\llparenthesis \mathbb{E}[\mathsf{throw}(c).\mathsf{catch}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[f(c)] \rrparenthesis_t & \text{(Catch)} \\
\llparenthesis \mathbb{E}[\mathsf{throw}(c).\mathsf{bind}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[\mathsf{throw}(c)] \rrparenthesis_t & \text{(Propagate-Error)} \\
\llparenthesis \mathbb{E}[A.\mathsf{fork}()] \rrparenthesis_t & \longrightarrow & \nu p.(\llparenthesis A \rrparenthesis^{\circ}_{t\cdot p} \mid \llparenthesis \mathbb{E}[\mathsf{pure}(t\cdot p)] \rrparenthesis_t), \;\; p \notin fn(\mathbb{E}, A) & \text{(Fork)} \\
\llparenthesis \mathbb{E}[\mathsf{lift}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[\mathsf{lift}(f)] \rrparenthesis^{\bullet}_t & \text{(Stuck-Async)} \\
\llparenthesis \mathbb{E}[\mathsf{lift}(f)] \rrparenthesis^{\bullet}_t & \xrightarrow{?c} & \llparenthesis \mathbb{E}[\mathsf{pure}(c)] \rrparenthesis_t & \text{(Async)} \\
\langle\rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{put}(V)] \rrparenthesis^{b}_t & \longrightarrow & \langle V\rangle_m \mid \llparenthesis \mathbb{E}[\mathsf{pure}(\mathsf{undef})] \rrparenthesis_t & \text{(Put-MVar)} \\
\langle V\rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{take}()] \rrparenthesis^{b}_t & \longrightarrow & \langle\rangle_m \mid \llparenthesis \mathbb{E}[\mathsf{pure}(V)] \rrparenthesis_t & \text{(Take-MVar)} \\
\langle V'\rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{put}(V)] \rrparenthesis_t & \longrightarrow & \langle V'\rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{put}(V)] \rrparenthesis^{\bullet}_t & \text{(Stuck-Put-MVar)} \\
\langle\rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{take}()] \rrparenthesis_t & \longrightarrow & \langle\rangle_m \mid \llparenthesis \mathbb{E}[m.\mathsf{take}()] \rrparenthesis^{\bullet}_t & \text{(Stuck-Take-MVar)}
\end{array}
$$

Figure 6. Transition rules for AsyncM values

$$
\begin{array}{rcll}
\llparenthesis \mathbb{F}[A.\mathsf{start}()] \rrparenthesis_u & \longrightarrow & \nu p.(\llparenthesis A \rrparenthesis^{\circ}_{p} \mid \llparenthesis \mathbb{F}[p] \rrparenthesis_u), \; p \notin fn(\mathbb{F}, A) & \text{(Start)} \\
\llparenthesis \mathbb{F}[\mathsf{new}\ MVar] \rrparenthesis_u & \longrightarrow & \nu m.(\langle\rangle_m \mid \llparenthesis \mathbb{F}[m] \rrparenthesis_u), \; m \notin fn(\mathbb{F}) & \text{(New-MVar)} \\
\llparenthesis \mathbb{F}[(x \Rightarrow M)(V)] \rrparenthesis_u, & \longrightarrow & \llparenthesis \mathbb{F}[M[V/x]] \rrparenthesis_u & \text{(Call)} \\
\llparenthesis \mathbb{F}[true\ ?\ M_1 : M_2] \rrparenthesis_u, & \longrightarrow & \llparenthesis \mathbb{F}[M_1] \rrparenthesis_u & \text{(True)} \\
\llparenthesis \mathbb{F}[false\ ?\ M_1 : M_2] \rrparenthesis_u, & \longrightarrow & \llparenthesis \mathbb{F}[M_2] \rrparenthesis_u & \text{(False)} \\
\llparenthesis A \rrparenthesis^{\circ}_t & \longrightarrow & \llparenthesis A \rrparenthesis_t & \text{(Run)} \\
\llparenthesis \mathsf{pure}(V) \rrparenthesis_t & \longrightarrow & \llparenthesis\rrparenthesis_t & \text{(Value-End)} \\
\llparenthesis \mathsf{throw}(c) \rrparenthesis_t & \longrightarrow & \llparenthesis\rrparenthesis_t & \text{(Error-End)} \\
\llparenthesis V \rrparenthesis_{\epsilon} & \longrightarrow & \llparenthesis\rrparenthesis_{\epsilon} & \text{(Main-End)} \\
\llparenthesis\rrparenthesis_u \mid P & \longrightarrow & P & \text{(GC)}
\end{array}
$$

Figure 7. Transition rules for terms, run thread, and termination

$$
\begin{array}{rcll}
\llparenthesis \mathbb{F}[t.\mathsf{cancel}()] \rrparenthesis_u & \longrightarrow & \llparenthesis \mathbb{F}[\mathsf{undef}] \rrparenthesis_u \mid \{\llbracket t_i \lightning\, \mathsf{cancel} \rrbracket\}_{\mathsf{prefix}(t,t_i)} & \text{(Cancel)} \\
\llparenthesis \mathbb{F}[t.\mathsf{pause}()] \rrparenthesis_u & \longrightarrow & \llparenthesis \mathbb{F}[\mathsf{undef}] \rrparenthesis_u \mid \{\llbracket t/t_i \lightning\, \mathsf{pause} \rrbracket\}_{\mathsf{prefix}(t,t_i)} & \text{(Pause)} \\
\llparenthesis \mathbb{F}[t.\mathsf{resume}()] \rrparenthesis_u & \longrightarrow & \llparenthesis \mathbb{F}[\mathsf{undef}] \rrparenthesis_u \mid \{\llbracket t/t_i \lightning\, \mathsf{resume} \rrbracket\}_{\mathsf{prefix}(t,t_i)} & \text{(Resume)} \\
\llbracket t \lightning\, \mathsf{cancel} \rrbracket \mid \llparenthesis \mathbb{E}[V] \rrparenthesis^{\bullet}_t & \longrightarrow & \llparenthesis \mathbb{E}[\mathsf{throw}(\text{“}interrupted\text{”})] \rrparenthesis_t & \text{(Cancel-Stuck)} \\
\llbracket t \lightning\, \mathsf{cancel} \rrbracket \mid \llparenthesis \mathbb{E}[\mathsf{lift}(f)] \rrparenthesis_t & \longrightarrow & \llparenthesis \mathbb{E}[\mathsf{throw}(\text{“}interrupted\text{”})] \rrparenthesis_t & \text{(Cancel-Async)} \\
\llbracket t/t_i \lightning\, \mathsf{pause} \rrbracket \mid \llparenthesis \mathbb{E}[\mathsf{lift}(f)] \rrparenthesis^{\bullet}_{t_i} & \xrightarrow{?c} & \llparenthesis \mathbb{E}[\mathsf{pure}(c)] \rrparenthesis^{\bullet,t}_{t_i} & \text{(Pause-Stuck)} \\
\llbracket t/t_i \lightning\, \mathsf{resume} \rrbracket \mid \llparenthesis \mathbb{E}[\mathsf{pure}(c)] \rrparenthesis^{\bullet,t}_{t_i} & \longrightarrow & \llparenthesis \mathbb{E}[\mathsf{pure}(c)] \rrparenthesis_{t_i} & \text{(Resume-Paused)}
\end{array}
$$

Figure 8. Transition rules for cancellation, pause, and resumption. Rule (Cancel-Stuck) has higher priority than Rule (Async).

`Progress` $p1$ and then runs the component threads with $p1$. When one of the threads wins the race, the call $p1$.cancel() terminates the rest of the threads.

```
// race :: [AsyncM a] -> AsyncM a
static race = lst => new AsyncM (p => {
  let p1 = new Progress(p);
  return Promise.race(lst.map(a => a.run(p1)))
    .finally(_ => {
        p1.cancel(); // cancel losing threads
        p1.unlink(); // remove p to p1 link
    });
})

// race :: [AsyncM a] -> AsyncM [a]
static all = lst => new AsyncM (p =>
    Promise.all(lst.map(a => a.run(p)))
)
```

Channel. We can use `MVar` to implement other primitives. For example, we have implemented a buffered channel using a `MVar` to hold the pending readers when the channel is empty and to hold the pending writers when the channel is full. Any threads that are blocked on a channel can be cancelled because they are blocked on its `MVar`.

Safe points. All asynchronous operations defined with *lift* are potential points of cancellation. An alternative is to run these operations using `Promise`s instead and to poll the cancellation status at specific check points. The *ifAlive* method below can be composed with other `AsyncM`s as a safe point for cancellation.

```
static ifAlive = new AsyncM (async p => {
  if (! p.cancelled) return;
  else throw("interrupted");
});
```

Control flow. Implementing control flow with `AsyncM` is more awkward than using *async*/*await*. For simple cases such as infinite loops, we can use methods like *loop* below.

```
// e.g. a.loop() runs forever unless cancelled
class AsyncM {
  loop = _ => new AsyncM (async p => {
      while (true) { await this.run(p) }
  })
}
```

For more complex control flow, it is better to compose the `AsyncM`s indirectly by first converting them to `Promise`s, which can be composed with `await`. The composed `Promise`s can then be converted back to an `AsyncM` object using an `async` function with a `Progress` parameter.

6 Evaluation

Our model is implemented with 400 lines of JavaScript. To evaluate its usability, we used it in two applications.

Data streaming. The first application (Figure 9) streams the voltage and current signals (sampled at 1KHz) from a remote source and visualizes them in two real-time charts. The voltage and current charts can be paused/resumed separately and can be stopped simultaneously.

To hide network latency, this application uses a thread (Thread A in Figure 10) to fetch data in batches at a regular interval and to push each data request to the request channel. Thread B reads a data request from the request channel, waits for it to complete, and writes its result to the voltage channel and the current channel using the `AsyncM.all` method. Thread V retrieves a batch of data from the voltage channel and displays the voltage data incrementally in a chart (e.g. by updating the chart with 10 new samples every 10 ms). Thread C performs similar actions for the current data.

All 4 threads run independently and communicate through the channels, which allows the speeds of data transmission and chart rendering to match. For example, if data transmission is faster than chart rendering, one or both of the data channels will become full, causing Thread B to block and the request channel to become full, which blocks Thread A. Also, we can pause the voltage (or the current) chart by pausing Thread V (or Thread C). Note that if Thread V is paused, the voltage channel will become full if Thread B keeps writing to it, which eventually blocks Thread B from writing to the current channel even if it is not full. Thus, if a chart is paused, Thread B may be cancelled and restarted so that it only writes data to the channel of the running chart.

Our thread abstractions help reduce the complexity of this application, where the channels manage thread synchronization, hierarchical thread cancellation allows all threads be cancelled as a group, and the ability to pause and resume threads makes it easy to pause and resume chart animations.

RxJS. The second application[3] is a subset of RxJS interface implemented with our thread library. RxJS[4] is a popular JavaScript library for reactive programming, which represents event streams as dataflow graphs. The core abstraction of RxJS is the `Observable` interface, which emits events to its observers connected through the `Subscription` objects. Despite its popularity, RxJS is difficult to debug [1] since, unlike the functional designs of reactive programming [6, 24], RxJS is imperative and its control flow does not correspond to its dataflow, which is hard to inspect with debuggers.

In our design, each `Observable` is implemented with a function that takes an `emitter` object and returns a thread that emits events to the emitter. As shown in Figure 11, when an `Observable` is subscribed, two threads are started: one is the `Observable` thread that emits events to an `emitter`, while the other thread calls a continuation k with the events received from the `emitter`. When the `Subscription` is unsubscribed, both threads are cancelled through a shared

[3]https://github.com/tianzhao/rxjs
[4]http://reactivex.io

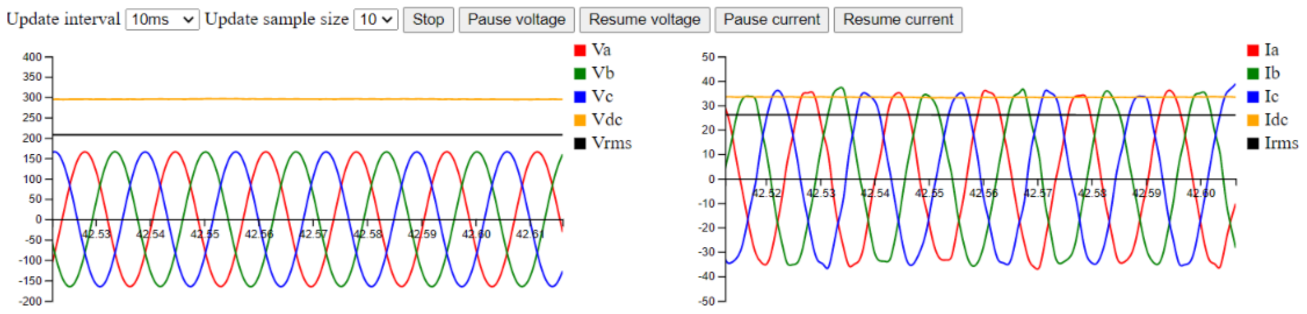

Figure 9. A data streaming application, where V_a, V_b, V_c, V_{dc}, V_{rms} are voltages and I_a, I_b, I_c, I_{dc}, I_{rms} are currents.

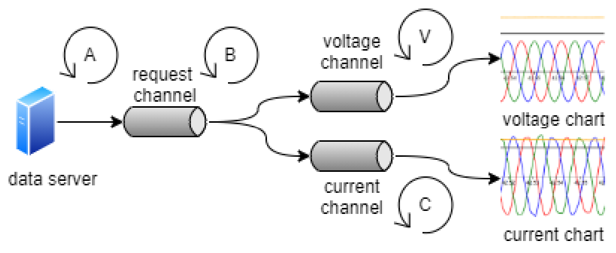

Figure 10. The architecture of the data streaming application in Figure 9, where A, B, C, and V are threads.

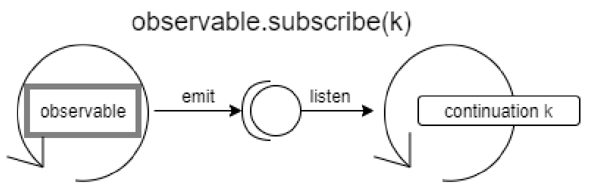

Figure 11. The subscription of a RxJS `Observable` using 2 threads (arrow circles) and an emitter (middle circle).

thread ID. Within an `Observable`, if an inner `Observable` is subscribed, its thread is the child of the parent `Observable`'s thread. Therefore, when an `Observable` is unsubscribed, its inner `Observables` are unsubscribed automatically due to the hierarchical cancellation of our thread model.

Our version of RxJS is easier to debug since the dataflow graph of a RxJS expression is embedded in the returned `Subscription` object, which contains an `emitter`, a thread ID, and the references to the `Subscriptions` to the inner `Observables`. Users can debug a RxJS program by navigating the `Subscription` object to examine the dataflow graph, inspecting its emitter for the past events, and checking its thread ID for the status of the subscription threads.

Runtime overhead. The most significant overhead in our design is due to the `AsyncM.lift` method, which allocates continuations and adds/removes cancellers. The table below shows in milliseconds the amount of time it takes to run a trivial synchronous and asynchronous computation (0s timeout) over a number of iterations. For synchronous computation, the overhead of `lift` is significant compared to `Promise`-based implementation. However, for asynchronous computation, the overhead due to `lift` is less noticeable.

iterations	Synchronous		Asynchronous	
	`AsyncM`	`Promise`	`AsyncM`	`Promise`
100	1.12	0.16	158.38	147.61
500	5.16	0.33	753.66	721.82
1000	5.42	0.55	1479.62	1458.18
5000	11.76	2.47	7314.38	7292.97
10000	17.99	4.92	14628.99	14590.47

7 Related Work

Promises. This work enhances JavaScript's `Promises` [5] by providing a thread-like concurrency model with cooperative cancellation. While `Promises` (together with `async` and `await`) allow us to write asynchronous programs in JavaScript in sequential style, the execution order of the programs is not always clear. JavaScript's event loop maintains separate queues for tasks (e.g. timeouts) and micro-tasks (e.g. `Promises`). `Promises` are executed in the order in which they are added to the micro-task queue. For example, the execution of two `Promise` chains can interleave even if they are all synchronous. Even if we call an `async` function without `await`ing for its result, its synchronous portion still runs first. Also, the `resolve` and `reject` functions of a `Promise` can be saved and invoked later by some other code, which can cause further interleaving of `Promise` execution. Consequently, a `Promise` chain may not have exclusive access to shared states in between asynchronous operations, which can lead to subtle race conditions. By wrapping `Promises` inside `AsyncMs`, we ensure that the threads must be explicitly started, the synchronous part of a thread has exclusive access to the shared states, and the locks such as `MVars` can be used to protect shared states between threads.

`Promises` also do not have builtin methods for cancellation but hand-crafted solution can easily forget pending callbacks or `Promises`, which leads to unintended side effects. Our proposal is intended to complement `Promises` by providing a more consistent way to terminate unused computation.

Coroutine. The exact definition for coroutines varies between languages, but they are generally understood as non-preemptive thread-like concepts [7, 12, 22]. They are non-preemptive in the way that control of execution can be suspended and transferred explicitly. A JavaScript generator is a limited form of coroutine that allows computation to switch back and forth between a generator and its caller through the `yield` mechanism. Together with `Promises`, generators can be used to compose asynchronous computation using `for..of` or `for await..of` loops. Python's *asyncio* library is an asynchronous framework that supports non-preemptive scheduling and also cancellation [9]. It provides low-level API that allows scheduling work from a different OS thread, in which case thread-safety needs to be considered.

Concurrency monad. Claessen [4] described the use of continuation monad for concurrency in Haskell. The design permits a limited form of concurrency on monadic computations without adding primitives to the language. The concurrency monad builds on the continuation monad, where computations can be evaluated concurrently by interleaving evaluation of lifted operations. By implementing it as a monadic transformer, the existing monads can be extended with concurrency operations and can be entirely defined as a library without introducing new language primitives. This idea was later adopted by Li and Zidancwic [13] in their scalable network services that provide type-safe abstractions for both events and threads. They use a continuation monad to build traces that are scheduled by the event loops.

Asynchronous Exception. Asynchronous exception is introduced in Concurrent Haskell [16], which allows one thread to throw an asynchronous exception to another thread. The asynchronous exception raises a synchronous exception in the receiving thread, which can be handled or cause the thread to terminate. Since Concurrent Haskell has preemptive scheduling, the asynchronous exception can interrupt the receiving thread at any point. To protect critical regions, Concurrent Haskell includes block and unblock primitives to mask the regions that cannot be interrupted. Despite this, threads blocked on `MVar` or IO can always be interrupted to reduce the chance of deadlock.

We adopt a similar strategy by throwing interrupt exception to target threads. However, since JavaScript is not preemptive, the interrupt exceptions are only received at the locations where the threads are blocked or polling the thread status. To disable the interrupt exceptions, we can run an `AsyncM` with a new `Progress` using the *block* method below.

```
class AsyncM {
  block = _ =>
    new AsyncM(_ => this.run(new Progress())) }
```

Our operational semantics is also modeled after that of Concurrent Haskell [16], where the reduction of processes is based on the chemical abstract machine [2, 3].

Cooperative Cancellation. AC [10] introduced language constructs to insert code blocks for asynchronous IO in native languages like C/C++. Each of these blocks is delimited by the `do..finish` keywords. Within a block, the keywords `async` and `cancel` can be used to start an asynchronous operation and to cancel it, respectively. The `cancel` keyword can be used with a label to indicate which `async` operation to cancel, but it must be used within the enclosing `do..finish` block. Cancelling an `async` operation will propagate cancellation into any nested `async` branches recursively. The execution of a `do..finish` block does not complete until all `async` operations within it are finished or cancelled.

.NET uses cancellation tokens for cooperative cancellation, where the concurrent tasks use their cancellation tokens to decide how to handle cancellation requests. The design distinguishes the cancellation source, which is used for requesting cancellation, from the tokens, which are used for polling cancellation status, registering cancellation callbacks, and enabling blocked tasks to wait for cancel events. A task can react to multiple cancellation tokens by linking them in a new cancellation source. *F#* [18, 23] provides an asynchronous programming model through CPS transformation on its `async` expressions. The language supports cancellation by implicitly threading cancellation tokens (derived from a cancellation source) through the program execution. Cancellation tokens are checked at IO primitives and various control flow constructs.

Our cancellation mechanism is similar to the cancellation tokens in that a thread reacts to the cancellation requests at some program points. However, we integrate cancellation into a thread model, where a thread ID is both the cancellation source and token. The hierarchical structure of threads allows a child thread to react to a cancellation request to its parent thread without explicitly linking cancellation tokens.

8 Conclusion

We have presented a thread-based concurrency model for JavaScript that can cancel, pause, and resume threads. The thread abstraction makes it easier to reason about asynchronous programs while synchronization primitives can protect shared resources. These advantages help reduce the occurrences of race conditions. The ability to cancel threads helps prevent the side effects of unwanted computation. The ability to pause and resume threads may be used for debugging concurrency errors in a browser environment and providing a simple way to suspend computation such as animation.

This design is implemented as a JavaScript library and since each `AsyncM` wraps a `Promise` function, it is compatible with the `Promise` abstraction and can be integrated with other types of programs using `async` and `await` to implement complex logic. The overhead of thread abstractions and cancellation is not significant relative to the computation time of the asynchronous operations that they support.

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)*. ACM, New York, NY, USA, 15–24. https://doi.org/10.1145/3427763.3428313

[2] Gérard Berry and Gérard Boudol. 1992. The Chemical Abstract Machine. *Theor. Comput. Sci.* 96, 1 (April 1992), 217–248. https://doi.org/10.1016/0304-3975(92)90185-I

[3] Gérard Boudol. 1992. Asynchrony and the Pi-Calculus.

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

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

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

[7] Ralf S. Engelschall. 2006. The GNU Portable Threads. https://www.gnu.org/software/pth/

[8] 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. https://doi.org/10.1109/ICST.2017.28

[9] Python Software Foundation. [n.d.]. The Python Language Reference. https://docs.python.org/3/reference/expressions.html

[10] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. 2011. AC: Composable Asynchronous IO for Native Languages. *SIGPLAN Not.* 46, 10 (Oct. 2011), 903920. https://doi.org/10.1145/2076021.2048134

[11] 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. https://doi.org/10.1109/ICST.2014.17

[12] D.E. Knuth. 2005. *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX – A RISC Computer for the New Millennium*. Pearson Education. https://books.google.com/books?id=imjwBQAAQBAJ

[13] 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

[14] Matthew C. Loring, Mark Marron, and Daan Leijen. 2017. Semantics of Asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages* (Vancouver, BC, Canada) *(DLS 2017)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/3133841.3133846

[15] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A Model for Reasoning About JavaScript Promises. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'17)*. ACM. https://doi.org/10.1145/3133910

[16] 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

[17] 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. https://doi.org/10.1145/2786805.2786820

[18] 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

[19] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. Race Detection for Web Applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 251–262. https://doi.org/10.1145/2254064.2254095

[20] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. 1996. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. ACM, New York, NY, USA, 295–308. https://doi.org/10.1145/237721.237794

[21] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, New York, NY, USA, 151–166. https://doi.org/10.1145/2509136.2509538

[22] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. 2010. Efficient Coroutines for the Java Platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java* (Vienna, Austria) *(PPPJ '10)*. ACM, New York, NY, USA, 20–28. https://doi.org/10.1145/1852761.1852765

[23] 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.

[24] 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)*. ACM, New York, NY, USA, 25–37. https://doi.org/10.1145/3427763.3428314