A Concurrency Model for JavaScript with Cooperative Cancellation

Anonymous Author(s)

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 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 monads 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.

ACM Reference Format:
Anonymous Author(s). 2021. A Concurrency Model for JavaScript with Cooperative Cancellation. In *Proceedings of Software Language Engineering (SLE '21).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [4] allowed the control flow of event callbacks be written in a more direct style that resembles sequential programs. A `Promise` is a builtin abstraction along with `async` and `await` keywords that allows asynchronous computation to be written in a sequential style. A `Promise` object, once constructed, starts immediately and upon completion, it either resolves successfully with a result or rejects with an error value.

It may seem unnecessary to provide a thread-based concurrency model for JavaScript since Promises already behave like threads with non-preemptive scheduling and methods for waiting 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 or resource, is paused, or is cancelled. A paused thread can be resumed or cancelled.

In this paper, we make the following contributions.

- We explain why a concurrency model with synchronization primitives and cancellation semantics is useful in JavaScript in Section 2.
- We propose a library-based design using the reader monad to represent thread-like abstractions in Section 4. The reader monad implicitly passes a thread ID throughout a computation so that it can be cancelled, paused, and resumed. A thread ID is a tree of progress objects that allows hierarchical cancellation. The implementation is available at https://anonymous.4open.science/r/thread-123E/AsyncM.js
- We define a primitive like Haskell's `MVar` and show how it supports communication and cancellation in Section 4.5 and how other primitives such as bounded buffer can be defined in Section 6.
- We give an operational semantics of our concurrency model in a style similar to that of Concurrent Haskell in Section 5.
- We discuss the usability and runtime overhead of this design in Section 7.

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,

SLE '21, 2021, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 idle time of event-loop for synchronous operations without blocking IO events (e.g. js-coroutine). 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 [9, 18, 20]. For example, event-race errors can be caused by concurrent access to an external resource (e.g. a web service) if it does not protect against concurrent 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 [6], inviting alternative methods to identify issues in such constructs.

While Promises [4] have helped many projects move away from deeply-nested callbacks, its semantics is still complex [12, 16] 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 [13] 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 a thread-based design has several advantages over Promises.

- 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 more well-defined abstraction boundaries and consistent semantics, 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 Promises and Concurrency Monad

Promises are similar in purpose to the continuation monads used for concurrency [3, 11], 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 must be run 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 */ })
```

The method *then* is similar to the bind operator >>= of a continuation monad. Exception handling of Promises is similar to stacking an *ExceptT* transformer on top of a continuation monad.

Our design is a cancellable concurrency monad with a thread ID. The thread ID is passed through the continuation monad by stacking a *ReaderT* transformer on top. In Haskell, the type of this monad could be written as

```
ExceptT Exception
        (ReaderT Progress (ContT () IO))
        a
```

where `Progress` is the type of thread ID and `a` is the type of result value. If we remove the type constructors, the actual type is quite simple.

```
Progress -> (Either Exception a -> IO ())
         -> IO ()
```

Since Promises already implements the concurrency monad except the thread ID, we can redefine it as:

```
Progress -> Promise a
```

which is a function that takes a progress object (i.e. thread ID) and returns a `Promise` that can resolve to a value of type `a`.

4 Concurrency with Cancellation

We define a JavaScript class `AsyncM` to represent our 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` lifts a value to an `AsyncM` that always return that value. The method `fmap` applies a function to `this` AsyncM while `bind` composes `this` AsyncM with a function that returns an AsyncM. These methods allow `AsyncM`s to be composed so that Progress value can be passed implicitly through the AsyncM computation. This style of composition does come with syntactic overhead since it prevents the use of `async` and `await` keywords for 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 canceller

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

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

An AsyncM that represents an asynchronous action is constructed with the `lift` method, whose parameter f performs asynchronous actions 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.

4.1 Thread ID and Cancellation

Thread IDs are used for cancellation. A thread ID is a `Progress` object, which is 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 = []
}
```

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 later
        return p
}}
```

When a lifted AsyncM runs on a `Progress` p, a canceller function is added to p. If the AsyncM completes, the canceller is removed. If it 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 from 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
```

Cancellation can happen anytime m is blocked, such as when running `ajax` or `display` function. However, like other non-preemptive designs, the timeout will not have immediate effects if it occurs while a synchronous action like `compute` is running.

Unlike Promises, which runs immediately after composition, the composition of AsyncM is separate from its execution, which helps identify the concurrent operations. One can certainly delay the execution of Promises 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.

4.2 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 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 static `lift` method checks whether its `Progress` p (i.e. the thread ID) is alive first. If it is not alive, then it returns a rejected `Promise` immediately. Otherwise, it adds a canceller c to the progress object p but c is removed from p if the `lifted` function f completes. The canceller c is called when p is cancelled.

```
class AsyncM {
  catch = h =>
      new AsyncM(p => this.run(p).catch(h))
}
```

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

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

4.3 Fork and Hierarchical Cancellation

Other than starting an independent thread, one can also fork a child thread with a progress object as 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, this reference is removed to avoid memory leak.

```
class AsyncM {
  fork = _ => new AsyncM (async p => {
      const p1 = new Progress(p)
      // unlink the reference from p to p1
      // after 'this' completes
      let f = _ => this.run(p1)
                   .finally(_ => p1.unlink())
      // start the thread asynchronously
      setTimeout(f, 0)
      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)
  }
}
```

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

```
class Progress {
  cancel = _ => {
    this.cancelled = true
    this.cancellers.forEach(
        // signal interrupts asynchronously
        canceller => setTimeout(canceller, 0)
    )
    this.children.forEach(
        child => child.cancel()
    )
    this.cancellers = []
  }
}
```

The interrupts are signaled asynchronously by running each canceller after a timeout. This is to be consistent with the semantics that a thread continues to run until it is blocked, paused, or cancelled.

Using `fork`, the AsyncM m of the previous example can be run 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 handles 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 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())
```

Lastly, we can use progress like a cancellation token. For example, one can start a group of threads by calling their `run` methods with the same progress t or the children of t so that any thread in the group can cancel the entire group through t.

4.4 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, add 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, a paused thread can be cancelled while 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 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, for example, a sibling thread of t that is responsible for resuming 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 above `lift` method 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
  pending = [] // pending 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 = _ => {
    this.paused = false
    let l = this.pending
    this.pending = []
    if (! this.cancelled)
        l.forEach(k => setTimeout(k, 0))
  }
}
```

The above `Progress` class has a pause status flag and a list of pending threads. 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.

4.5 Synchronization Mechanism

In JavaScript, event races can be caused by 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, consider a real-life bug[1], where button clicks send requests to a remote service but the service has no support for concurrent requests. If user clicks on the button again before the previous one completes, the next request will cause an internal error in the service.

The code below represents 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.

```
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 to the progress object 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 removes the thread from the list of threads pending on the `MVar` and throws 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 = _ => { // canceller
        this.pending =
            this.pending.filter(t => t != k)
        reject("interrupted"); // 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()()
  }

  // take method is omitted
}
```

The `take` method is omitted, which registers a canceller (identical to that of the `put` method) with the progress *p* if the `MVar` is empty and the canceller will be removed when the `MVar` is filled.

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*. Since JavaScript is not preemptive, however, it is less likely to enter a deadlock state due to non-determinism than a language with preemptive scheduling.

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

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

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

Figure 1. The syntax of AsyncM, values, and terms

5 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 1, 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 pure(V) that returns value V, throw(c) that throws an error c, and lift(f) that runs asynchronous function f and waits for its results. AsyncM also includes combinators: A.bind(f) that passes the value of A to f, A.catch(f) that catches the error of A with f, and A.fork() that runs A in a child thread.

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.

P, Q, R	::=		States
	|	$(\!\lvert M \rvert\!)_u$	live thread with ID u
	|	$(\!\lvert M \rvert\!)^\bullet_t$	stuck thread
	|	$(\!\lvert M \rvert\!)^\circ_t$	ready thread
	|	$(\!\lvert \rvert\!)_u$	completed thread
	|	$\langle\rangle_m$	empty `MVar` named m
	|	$\langle V \rangle_m$	`MVar` m filled with V
	|	$[\![t \lightning \text{cancel}]\!]$	t is cancelled
	|	$[\![t/t_i \lightning \text{pause}]\!]$	t_i is paused by t
	|	$[\![t/t_i \lightning \text{resume}]\!]$	t_i is resumed by t
	|	$\nu x.P$	restriction
	|	$P \mid Q$	parallel composition

Figure 2. The syntax of program states

5.1 Program Transitions

We define our semantics by describing the transitions between program states. A program state (Figure 2) is a parallel composition of threads, `MVars`, and the cancel/pause/resume actions on threads.

A thread is either alive $(\!\lvert M \rvert\!)_u$, stuck $(\!\lvert M \rvert\!)^\bullet_t$, or ready $(\!\lvert M \rvert\!)^\circ_t$, where the subscript is the thread ID. A live thread is currently running, the stuck threads are waiting for events or blocked on `MVars`, and a ready thread will run when there is no other live thread. The initial program state is the main thread $(\!\lvert M \rvert\!)_\epsilon$.

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 program state is supported by the equivalence relation $\equiv$ defined in Figure 3 – identical to the one in Concurrent Haskell [15, 19].

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

Figure 3. Structural congruence

$$
\mathsf{live}(\langle\!|M|\!\rangle_u)\ \frac{\mathsf{live}(P)}{\mathsf{live}(vx.P)}\ \frac{\mathsf{live}(P)}{\mathsf{live}(P \mid Q)}\ \frac{\mathsf{live}(Q)}{\mathsf{live}(P \mid Q)}
$$

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

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

Figure 4. Structural transitions

The structural transitions of program states are defined in Figure 4. 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.

5.2 Transition Rules

We first explain the transition rules for AsyncM values (Figure 5) and then explain the rules for terms (Figure 6) and the rules for the actions on threads (Figure 7). The transition rules for AsyncM values describe the thread computation, which take 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. Rule (Async) and (Stuck-Async) describe how $\mathsf{lift}(f)$ 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 $\langle\!|\mathbb{E}[\mathsf{lift}(f)]|\!\rangle_t^{\bullet}$ transitions to a state that returns the event value c.

MVar. Rule (Put-MVar) says $\langle\!|\mathbb{E}[m.\mathsf{put}(V)]|\!\rangle_t^b$ 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 (Figure 6) take place within the context $\mathbb{F}$ defined below.

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

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 4, 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 are stuck. Figure 6 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 7 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 $\{[\![t_i \not\downarrow \mathsf{cancel}]\!]\}_{\mathsf{prefix}(t,t_i)}$. 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 and the children of t. By Rule (Cancel-Stuck), the cancel action will cause a stuck thread to throw an exception. By Rule (Cancel-Async), the cancel action will cause a live

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

Figure 5. Transition rules for AsyncM values

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

Figure 6. Transition rules for terms, run thread, and termination

thread with an asynchronous operation to throw an exception. The latter case only occurs when a thread is cancelling itself.

Rule (Pause) describes how $t.\mathsf{pause}()$ reduces to the undef value in its context and produces $\{\!|t/t_i \not\downarrow \mathsf{pause}|\!\}_{\mathsf{prefix}(t,t_i)}$. By Rule (Pause-Stuck), each of the actions can pause a stuck thread after 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 with the call $t.\mathsf{resume}()$ according to Rules (Resume) and (Resume-Paused).

6 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 in parallel and 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 progress $p1$ and then runs the component threads with $p1$. When one of the threads wins the race, the call $p1.\mathsf{cancel}()$ 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
```

$$
\begin{array}{rcll}
(\!|\mathbb{F}[t.\mathsf{cancel}()]|\!)_u & \longrightarrow & (\!|\mathbb{F}[\mathsf{undef}]|\!)_u \mid \{[\![t_i \not\downarrow \mathsf{cancel}]\!]\}_{\mathrm{prefix}(t,t_i)} & \text{(Cancel)} \\
(\!|\mathbb{F}[t.\mathsf{pause}()]|\!)_u & \longrightarrow & (\!|\mathbb{F}[\mathsf{undef}]|\!)_u \mid \{[\![t/t_i \not\downarrow \mathsf{pause}]\!]\}_{\mathrm{prefix}(t,t_i)} & \text{(Pause)} \\
(\!|\mathbb{F}[t.\mathsf{resume}()]|\!)_u & \longrightarrow & (\!|\mathbb{F}[\mathsf{undef}]|\!)_u \mid \{[\![t/t_i \not\downarrow \mathsf{resume}]\!]\}_{\mathrm{prefix}(t,t_i)} & \text{(Resume)} \\
[\![t \not\downarrow \mathsf{cancel}]\!] \mid (\!|\mathbb{E}[V]|\!)^{\bullet}_t & \longrightarrow & (\!|\mathbb{E}[\mathsf{throw}(\text{“}\mathit{interrupted}\text{”})]|\!)_t & \text{(Cancel-Stuck)} \\
[\![t \not\downarrow \mathsf{cancel}]\!] \mid (\!|\mathbb{E}[\mathsf{lift}(f)]|\!)_t & \longrightarrow & (\!|\mathbb{E}[\mathsf{throw}(\text{“}\mathit{interrupted}\text{”})]|\!)_t & \text{(Cancel-Async)} \\
[\![t/t_i \not\downarrow \mathsf{pause}]\!] \mid (\!|\mathbb{E}[\mathsf{lift}(f)]|\!)^{\bullet}_{t_i} & \xrightarrow{?c} & (\!|\mathbb{E}[\mathsf{pure}(c)]|\!)^{\bullet;t}_{t_i} & \text{(Pause-Stuck)} \\
[\![t/t_i \not\downarrow \mathsf{resume}]\!] \mid (\!|\mathbb{E}[\mathsf{pure}(c)]|\!)^{\bullet;t}_{t_i} & \longrightarrow & (\!|\mathbb{E}[\mathsf{pure}(c)]|\!)_{t_i} & \text{(Resume-Paused)}
\end{array}
$$

Figure 7. Transition rules for cancellation, pause, and resumption

```
        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 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 as Promises instead and to poll the cancellation status at specific check points. The *ifAlive* method below can be composed with other AsyncMs as a safe point for cancellation.

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

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

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

To implement more complex control flow, it is easier to define an AsyncM object that encloses an async function that takes a Progress parameter.

7 Evaluation

Our model is implemented with 500 lines of JavaScript. To evaluate its usability, we use it in two applications.

Data streaming. The first application (Figure 8) is a data streaming program that visualizes electrical signals sampled at 1KHz from a remote source. The application uses a fetch thread that sends an Ajax request every second to a server for 1000 samples and pushes the future response to a request channel. It uses a processing thread that reads the future response from the request channel, waits for it to complete, and pushes the results into a voltage data channel and a current data channel. Two display threads are used to retrieve voltage and current from the data channels and to display the data incrementally in two charts (implemented with the library `d3.js`). The chart update is wrapped in a `Promise`, which is converted to an `AsyncM` that runs in a regular interval. This application includes menus to change the update interval (i.e duration) and update size (i.e. stride). For example, with the duration of 10 ms and the stride of size 10, the chart will refresh every 10 ms with 10 new samples (the chart holds 100 samples). The menus and buttons are implemented as threads as well. All threads in this application can be cancelled altogether through the stop button. The voltage and current charts can be paused and resumed independently since the chart updates are separate threads. Through the channels, the application can fetch and display data in separate threads so that the charts are updated smoothly without the latency of the Ajax calls.

RxJS. We also implemented a small subset of RxJS using our thread model[2]. RxJS is a popular JavaScript library for reactive programming, which handles asynchronous events through dataflow graphs. The core concept of RxJS is the `Observable` object, which emits events to its observers connected through `Subscriptions`. Despite its popularity, RxJS is difficult to debug since its dataflow graphs are hard to

[2] https://anonymous.4open.science/r/rxjs-83BD/rxjs.js

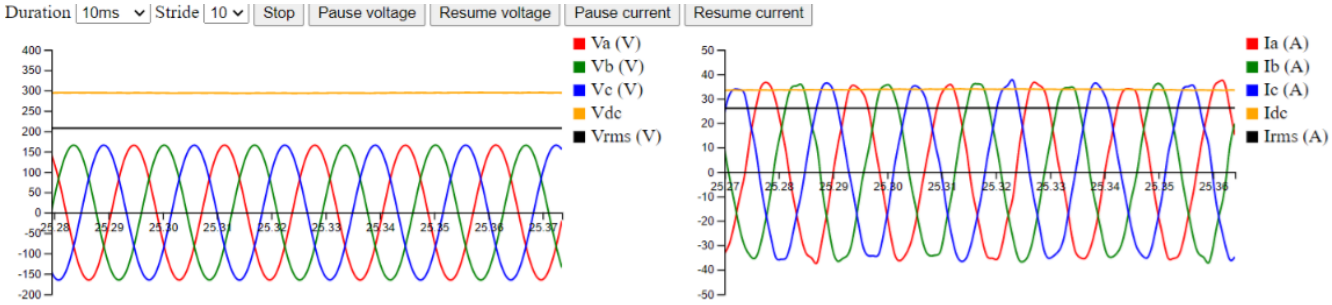

Figure 8. A data streaming application, where V_a, V_b, V_c, V_{dc}, V_{rms} are voltages and I_a, I_b, I_c, I_{dc}, I_{rms} are currents.

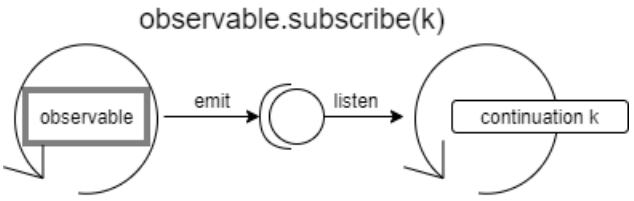

Figure 9. The subscription of a RxJS `Observable` using 2 threads (arrow circles) and an emitter (middle circle).

inspect through debugging tools. Using our thread abstraction, the dataflow logic of RxJS is simplified. For example, we implemented 24 operators of RxJS with 600 lines of code including comments and spaces. Figure 9 illustrates how the `Subscription` of an `Observable` is implemented using two threads: one runs the `Observable` and emits its events to an `emitter`, while the other runs the continuation k for each event received from the `emitter`. With our design, the dataflow graph of a RxJS program is easy to inspect, which is embedded in the `Subscription` objects that contain `emitters`, thread IDs, and references to the related `Subscriptions`. Users can debug RxJS programs by inspecting the emitters for past events and checking the thread IDs 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 negligible.

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

8 Related Work

Promises. This work enhances JavaScript's Promises [4] 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 resulting program is not always clear. JavaScript's event loop maintains separate queues for tasks such as timeouts and micro-tasks such as 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. When calling an `async` function, its synchronous portion runs first with or without `awaiting` for its result. 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 AsyncM, we ensure that a thread must be explicitly started and while a thread is running its synchronous code, it has exclusive access to shared states. Locks in form of `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 [5, 10, 21]. 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 [7]. 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 [3] described the use of continuation monad for concurrency in the context of 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, 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 [11] in their scalable network services that provides type-safe abstractions for both events and threads. They use a continuation monad to build traces that are scheduled by event loops.

Another work based on the continuation monad [14] utilizes the *applicative* abstraction in Haskell, which is more general than monad. This design does not have an explicit fork operation but can provide concurrency implicitly through its combinators. It benefits other operations that are based this abstraction, automatically providing concurrency to them.

Asynchronous Exception. Asynchronous exception is introduced in Concurrent Haskell [15], 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. If we want to disable interruption for an `AsyncM` computation, we can run it 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 [15], where the reduction of processes is based on the chemical abstract machine [1, 2].

Cooperative Cancellation. AC [8] 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 operation concurrently and to cancel them, respectively. The `cancel` keyword can be used with a label to indicate which `async` work to cancel, but it must be used within the enclosing `do..finish` block. Cancelling an `async` work will propagate cancellation into any nested `async` branches within it recursively. The execution of a `do..finish` block will only move on until all `async` work within it has finished or cancelled.

F# [17, 22] provides an asynchronous programming model through CPS transformation on the async expressions. The language supports cancellation by implicitly threading cancellation tokens (derived from a cancellation source) through the execution of computational effect. Cancellation tokens are checked at IO primitives and various control flow constructs and the cancellation effect is not necessarily immediate. Cancellation tokens are also used in .NET[3] for cooperative cancellation, where 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.

Our cancellation mechanism is similar to cancellation tokens in that a thread (or a task) reacts to 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 cancellation requests to its parent threads without explicitly linking cancellation tokens.

9 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 pause and resume threads also helps with debugging concurrency errors in a browser environment and provides an easy way to suspend computation such as animation. The ability to cancel threads helps reduce the side effects of unwanted computation.

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

[3] https://docs.microsoft.com/en-us/dotnet/standard/threading/cancellation-in-managed-threads

References

[1] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theor. Comput. Sci.*, 96(1):217–248, April 1992.

[2] Gérard Boudol. Asynchrony and the pi-calculus, 1992.

[3] Koen Claessen. A poor man's concurrency monad. *J. Funct. Program.*, 9(3):313–323, May 1999.

[4] ECMA International. ECMA-262: ECMAScript 2015 language specification. Standard, ECMA International, June 2015.

[5] Ralf S. Engelschall. The gnu portable threads, Jun 2006.

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

[7] Python Software Foundation. The python language reference.

[8] Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. AC: Composable asynchronous io for native languages. *SIGPLAN Not.*, 46(10):903920, October 2011.

[9] Shin Hong, Yongbae Park, and Moonzoo Kim. Detecting concurrency errors in client-side JavaScript web applications. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 61–70. IEEE, 2014.

[10] D.E. Knuth. *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX – A RISC Computer for the New Millennium*. Pearson Education, 2005.

[11] Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *SIGPLAN Not.*, 42(6):189–199, June 2007.

[12] Matthew C. Loring, Mark Marron, and Daan Leijen. Semantics of asynchronous JavaScript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, DLS 2017, pages 51–62, New York, NY, USA, 2017. ACM.

[13] Magnus Madsen, Ondřej Lhoták, and Frank Tip. A model for reasoning about JavaScript promises. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):86, 2017.

[14] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. *SIGPLAN Not.*, 49(9):325–337, August 2014.

[15] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in haskell. *SIGPLAN Not.*, 36(5):274–285, May 2001.

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

[17] Tomas Petricek and Don Syme. The F# computation expression zoo. In *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages - Volume 8324*, PADL 2014, page 33–48, Berlin, Heidelberg, 2014. Springer-Verlag.

[18] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262, New York, NY, USA, 2012. ACM.

[19] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 295–308, New York, NY, USA, 1996. Association for Computing Machinery.

[20] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.

[21] Lukas Stadler, Thomas Würthinger, and Christian Wimmer. Efficient coroutines for the java platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, page 20–28, New York, NY, USA, 2010. Association for Computing Machinery.

[22] Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages*, pages 175–189, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.