December 2018

Concurrency Analysis in Javascript Programs Using Arrows

Joshua Cochrane
University of Wisconsin-Milwaukee

CONCURRENCY ANALYSIS IN JAVASCRIPT PROGRAMS USING ARROWS

by

Josh Cochrane

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Computer Science

at

The University of Wisconsin-Milwaukee

December 2018

ABSTRACT

CONCURRENCY ANALYSIS IN JAVASCRIPT PROGRAMS USING ARROWS

by

Josh Cochrane

The University of Wisconsin-Milwaukee, 2018
Under the Supervision of Professor Tian Zhao

Concurrency errors are difficult to detect and correct in asynchronous programs such as those implemented in JavaScript. One reason is that it is often difficult to keep track of which parts of the program may execute in parallel and potentially share resources in unexpected, and perhaps unintended, ways. While programming constructs such as promises can help improve the readability of asynchronous JavaScript programs that were traditionally written using callbacks, there are no static tools to identify asynchronous functions that run in parallel, which may potentially cause concurrency errors.

In this work, we present a solution for implementing JavaScript programs using a library based on the abstraction of arrows. We enhanced the previous implementation of the arrows library by enabling its use with Node.js and by adding parallel asynchronous path detection. Automated identification of which arrows may execute in parallel helps the programmer narrow down the possible sources of concurrency errors.

To

my parents,

my wife,

and my children,

who have supported me

through this endeavor

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

1 Introduction

Concurrency errors in asynchronous programs are notoriously difficult to detect and correct due to the difficulty in reproducing the errors and the fact that such errors are often caused by mistakes in other parts of the program. 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, which results in unexpected effects. To detect errors caused by event races, programmers have to identify the culprits of the errors, which often are event handlers that can run in an unexpected order.

1.1 Motivation

For example, a real-life concurrency error reported on GitHub [1] involves an application which, at the time, only supported one user in the database. When a developer clicks the button to create the first user, the event handler first checks if the database already has a user before it starts the asynchronous creation process, which then takes some time to complete. Meanwhile, there is no feedback to the user that it is in progress, so in some cases the developer would click the button and trigger the event again. If the first creation had not finished, the database would still be empty and the initial check would pass again, so it would start creating another user. When the second user was added to the database, it would report uniqueness constraint violations and cause other unspecified problems.

We replicate the implementation in question with a simplified code snippet below, where `createOneUser` is an asynchronous function that communicates with a remote database to confirm that it is empty, creates and adds a user to the database, and then activates its callback on completion.

```
button.on('click', () => createOneUser(callback));
```

Listing 1: A simple handler `createOneUser`

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

If the user clicks the button twice, the button event handler will run twice. The initial database-check will pass, and if the first call to `createOneUser` takes a long time to complete, the database-check in the second instances will pass as well. As a result, two users are created in the database. This is an event race because multiple instances of the handler to the button events can run in parallel while the asynchronous calls within the event handler cannot prevent race conditions.

Note that it may have been true when the code was first written that the handler would complete well before the user could reasonably click the button again, but (based on the issue report), stronger password hashing algorithm usage or execution on a slower platform appears to have broken that assumption, making it easier for a user to trigger this behavior.

Once we identify the source of the error, fixing it is rather easy, at least naively. Below is a corrected version, which uses a local variable `active` to disable the button listener before a call to `createOneUser` returns. Using 'ad-hoc' synchronization like this is not fool-proof, but does add a layer of protection in this case.

```
let active = false;
button.on('click', () => {
  if (!active) {
    active = true;
    createOneUser(() => {active = false; callback()});
  }
});
```

Listing 2: `createOneUser` protected with ad-hoc synchronization

The question, then, is how to identify the potential event races like this automatically, which is necessary for analyzing larger applications. Analyzing JavaScript programs with dependencies on multiple libraries is very difficult. Despite advances in program analysis techniques and algorithms for asynchronous JavaScript programs, statically detecting event races may require static analysis of the entire JavaScript language [12], which can be imprecise and time consuming, particularly for such a complex and dynamic language.

Promises, introduced in ECMAScript 6, are a language construct provided to help describe and control asynchronous execution. While they have helped many projects move

away from deeply-nested callbacks (colloquially known as 'callback hell'), they are not without their own problems. The semantics are still complex, and the number and breadth of issues reported on platforms like Stack Overflow indicate users still struggle to understand their proper use. Not many static tools exist to help analyze the specific behavior of promises to identify potential issues, and none we have found help identify concurrency issues amongst promises. Madsen et al. [10] presented the Promise Graph, which helps track when promises are defined and activated/resolved as a step toward helping developers identify issues, but it still does not indicate when pieces may execute in parallel and may cause concurrency errors. They argue that still more could be done to help programmers ensure the correctness of their promise-based code.

Static analysis of race conditions and concurrency errors in JavaScript is also fundamentally limited by the complexity and dynamic nature of the language itself [12, 6, 11]. The 'eval' construct, which can (for instance) be used to run arbitrary user-provided code, could not be ignored for full coverage. Even if you skip support for that impossibly-dynamic construct, the specifics of the happens-before relationship for event handlers often cannot be determined statically, as it may depend on external events.

1.2 Our Contributions

In this work, we propose a lightweight implementation of asynchronous JavaScript programs using the arrows library [3], where the asynchronous computation can be analyzed before it starts. An arrows program encodes programming logic with compositions of arrows, which are lifted functions that perform synchronous or asynchronous computation. An arrows program executes in two stages: a composition stage and an execution stage. After the composition stage, we can analyze the structure of the composed arrows and determine the possible pairs of asynchronous arrows that are potentially parallel.

An arrows-based implementation of the above example is shown below, where `Arrow.fix` introduces a recursive arrow in which `a` is a variable that refers to the recursive arrow itself.

```
// Arrow Syntax
var x = Arrow.fix(a => button.on('click',
            a.any(createOneUser.seq(callback).seq(a))
        )
);
x.run();
```

Listing 3: A simple arrows implementation of `createOneUser`.

The call `.on` binds an arrow once to the named event on the referenced arrow (`button`) while `a.any(createOneUser.seq(callback).seq(a))` is an `any` arrow that races two parallel arrows: `a` and `createOneUser.seq(callback).seq(a)`. In an `any` arrow, progress made by one arrow will cancel the other arrows in that race. In this case, either `createOneUser` makes progress, in which case the parallel arrow `a` is canceled, or arrow `a` makes progress when the button is clicked again, in which case the `createOneUser` is canceled. If `createOneUser` makes progress, then the sequenced arrow `callback` will run, which is followed by arrow `a` again. Note that an arrow registered to an event source is immediately deregistered once an event occurs and the arrow is called, or if the arrow is canceled. Hence, we use recursion to implement the click event listener.

The arrow `x` does not run until we call its `run` method, which provides an opportunity for us to analyze potential event races in `x` before its actual execution. A simple analysis of `x` does reveal that there is a race condition between the arrow `createOneUser` and *itself*. That is, in the `any` arrow, users can click the button again before `createOneUser` returns, in which case another instance of `createOneUser` is launched.

If user clicks button again before `createOneUser` returns, then `createOneUser` is canceled in the sense that its result is ignored and its subsequent sequenced arrows are not executed. Normally, this would be a good implementation if `createOneUser` does not have harmful side effect to the remote server, as it allows users to cancel an operation that takes too long. An example would be to 'cancel' the loading of an image from a server; while we may not be able to cancel the request for the image outright, we can safely just ignore the image returned.

In this case, the implementation does present an event race, as the asynchronous operation

eventually modifies a common database. Once a user has noticed errors when running this code, the user can analyze the arrows code to determine arrows that may run in parallel and use the results to identify the source of the error, which in this case is the simultaneous execution of multiple instances of `createOneUser`.

To fix this problem, we can make a simple change as follows, which is to remove the `any` arrow combinator so that users cannot preemptively cancel an execution of `createOneUser` and this arrow must execute one at a time.

```
var x = Arrow.fix(a => button.on('click', createOneUser)
                        .seq(callback).seq(a)
);
x.run();
```

Listing 4: A safer arrows implementation of `createOneUser`.

The button click will have no effect because each click event will cause `createOneUser` be deregistered from the button and it will not be registered again until the previous call returns and subsequent computation completes. An analysis can reveal that there is no parallel execution of arrows within this code.

In summary, we make the following contributions:

- We extend and update the existing arrows library (syntax, semantics, and implementation) to support lifting of asynchronous functions, and enable its use with Node.js applications.
- We present rules for parallel arrow detection and show its application in detecting concurrency errors in arrow programs.
- We discuss real-world use cases demonstrating the usefulness of parallel arrow analysis.

For the rest of the paper, we first give a formal definition of the arrows library in Section 2, which includes its syntax and semantics. In Section 3, we present the rules for detecting asynchronous arrows that may run in parallel. In Section 4, we discuss the implementation of case studies to use arrows to detect event races. Related works are discussed in Section 5. In Section 6, we conclude and suggest directions for future research.

2 Arrows library for JavaScript

In this section, we discuss the formal syntax and semantics of arrows. The definitions are similar to those in [3] with some updates but we included the entire set of definitions for clarity.

2.1 Syntax of Arrows

An arrow is either constructed with a function or composed through combinators. The arrow abstraction supports the implementation of asynchronous JavaScript programs with a sequential style similar to JavaScript promises. The major difference between arrows and promises for our purpose is that the execution of an arrow a is delayed until $a.\mathsf{run}()$ is called. This gives us an opportunity to analyze the structure of the arrow before it actually runs.

As shown in Figure 1, arrows are constructed through two constructors: lift, which lifts a synchronous function to an arrow, and alift, which lift an asynchronous function. Arrows a_1 and a_2 can be sequenced with seq combinator as in $a_1.\mathsf{seq}(a_2)$. They can also run in parallel with all or any combinators where $a_1.\mathsf{any}(a_2)$ completes either arrow a_1 or arrow a_2 while $a_1.\mathsf{all}(a_2)$ completes both arrows a_1 and a_2. In $a_1.\mathsf{any}(a_2)$, if one of the two arrows makes progress, then the other one is cancelled. An arrow lifted from a function f makes progress if f completes. A composite arrow makes progress if any part of the arrow makes progress. The progress of an arrow a may be suppressed with $\mathsf{noemit}(a)$, so that it only makes progress when arrow a completes. For example, in $(a_1.\mathsf{seq}(a_2)).\mathsf{any}(a_3)$, the progress of arrow a_1 will cancel arrow a_3, while in $\mathsf{noemit}(a_1.\mathsf{seq}(a_2)).\mathsf{any}(a_3)$, only the completion of $a_1.\mathsf{seq}(a_2)$ can cancel arrow a_3. The try combinator is used to handle exceptions that may be thrown during the execution of an arrow. For example, in the arrow $a_1.\mathsf{try}(a_2, a_3)$, if arrow a_1 completes successfully, its result will be passed to the arrow a_2, and if arrow a_1 throws an exception, then the exception will be handled by the arrow a_3. In the arrow $a_1.\mathsf{on}(s, a_2)$, the arrow a_2 runs only if the event s occurs on an event emitter returned by arrow a_1.

$a \in Arrow$	::=	$\mathsf{lift}(f)$	lift a synchronous function as arrow
	|	$\mathsf{alift}(f)$	lift an asynchronous function as arrow
	|	$a_1.\mathsf{on}(s, a_2)$	event handling
	|	$a_1.\mathsf{seq}(a_2)$	arrow sequence
	|	$a_1.\mathsf{all}(a_2)$	all arrow
	|	$a_1.\mathsf{any}(a_2)$	any arrow
	|	$a_1.\mathsf{try}(a_2, a_3)$	try arrow
	|	$\mathsf{noemit}(a)$	no emit arrow
	|	$\mathsf{fix}(\alpha \Rightarrow a)$	recursive arrow
f	::=	$\lambda x.e$	function
s			string expression
$e \in Exp$	::=	...	
	|	$a.\mathsf{run}()$	run arrow

Figure 1: Syntax of $\lambda_{\text{asyncArrow}}$.

2.2 Semantics of Arrows

The arrows are translated using the rules in Figure 2 to functions of the form $\lambda x.p.k.h.\ e$, where the parameter x is the arrow input, the parameter p is a progress value, the parameter k is a continuation function for normal execution, and the parameter h is a continuation function to handle exceptions.

An arrow lifted from a synchronous function f is translated to a function that applies continuation k to the result of the call to f if it returns normally or in case of an exception, applies the error handler h to the exception. An arrow lifted from an asynchronous function f is translated to an async expression of the form $\mathsf{async}\ v_e\ p\ \lambda z.e$ that registers the pair $(p, \lambda z.e)$ with the event value v_e, which is written as $\mathsf{event}(f, x)$. If the event fires, the callback $\lambda z.e$ is invoked. If z is a success, the callback signals the progress of the arrow with $\mathsf{adv}\ p$ (which may cancel other arrows related to p) and calls the continuation k.

The arrow $a_1.\mathsf{on}(s, a_2)$ runs a_2 if the event emitter y returned from a_1 has an event of the name s, which is represented by the event value $\mathsf{event}(y, s)$. The arrow $a_1.\mathsf{seq}(a_2)$ sequences

$$
\begin{array}{rcll}
[\![\mathsf{lift}(f)]\!] & \equiv & \lambda x.p.k.h. & \mathsf{case}\ (f\ x)\ \mathsf{of}\ \mathrm{succ}(y) \Rightarrow k\ y,\ \mathrm{fail}(y) \Rightarrow h\ y \\
[\![\mathsf{alift}(f)]\!] & \equiv & \lambda x.p.k.h. & \mathsf{async}\ \mathsf{event}(f, x)\ p\ \lambda z.\ \mathsf{case}\ z\ \mathsf{of}\ \begin{array}{l}\mathrm{succ}(y) \Rightarrow \mathsf{adv}\ p;\ k\ y, \\ \mathrm{fail}(y) \Rightarrow h\ y\end{array} \\
[\![a_1.\mathsf{on}(s, a_2)]\!] & \equiv & \lambda x.p.k.h. & [\![a_1]\!]\ x\ p\ (\lambda y.\ \mathsf{async}\ \mathsf{event}(y, s)\ p\ \lambda z.[\![a_2]\!]\ z\ p\ k\ h)\ h \\
[\![a_1.\mathsf{seq}(a_2)]\!] & \equiv & \lambda x.p.k.h. & [\![a_1]\!]\ x\ p\ (\lambda y.[\![a_2]\!]\ y\ p\ k\ h)\ h \\
[\![a_1.\mathsf{all}(a_2)]\!] & \equiv & \lambda x.p.k.h. & y = [(), ()]; \\
 & & & [\![a_1]\!]\ x[1]\ p\ (\lambda z.\ y[1] = z;\ \mathsf{if}\ y[2] \neq ()\ \mathsf{then}\ k\ y)\ h; \\
 & & & [\![a_2]\!]\ x[2]\ p\ (\lambda z.\ y[2] = z;\ \mathsf{if}\ y[1] \neq ()\ \mathsf{then}\ k\ y)\ h \\
[\![a_1.\mathsf{any}(a_2)]\!] & \equiv & \lambda x.p.k.h. & [\![a_1]\!]\ x\ (P_i^1 :: p)\ k\ h;\ [\![a_2]\!]\ x\ (P_i^2 :: p)\ k\ h \\
[\![a_1.\mathsf{try}(a_2, a_3)]\!] & \equiv & \lambda x.p.k.h. & [\![a_1]\!]\ x\ (P_i^1 :: p) \\
 & & & \lambda y.\ [\![a_2]\!]\ y\ p\ k\ h \\
 & & & \lambda y.\ \mathsf{adv}\ (P_i^2 :: \epsilon);\ [\![a_3]\!]\ y\ m\ k\ h \\
[\![\mathsf{noemit}(a)]\!] & \equiv & \lambda x.p.k.h. & [\![a]\!]\ x\ (Q :: p)\ (\lambda y.\ \mathsf{adv}\ p;\ k\ y)\ h \\
[\![\mathsf{fix}(\alpha \Rightarrow a)]\!] & \equiv & \mathsf{fix}(\lambda\alpha. & [\![a]\!]) \\
[\![\alpha]\!] & \equiv & \alpha & \\
[\![a.\mathsf{run}()]\!] & \equiv & [\![a]\!] & ()\ \epsilon\ \lambda x.x\ \lambda x.x
\end{array}
$$

Figure 2: Arrow translation rules

the execution of a_1 and a_2 by passing the computation of a_2 as a continuation to a_1. The arrow $a_1.\mathsf{all}(a_2)$ runs both a_1 and a_2 in parallel by using a shared array y to store the arrow output so that when both arrows complete, the continuation to the arrow k will be applied to y. The arrow $a_1.\mathsf{any}(a_2)$ makes a_1 and a_2 race by calling $[\![a_1]\!]$ and $[\![a_2]\!]$ with a pair of progress value $P_i^1 :: p$ and $P_i^2 :: p$ so that the progress of one arrow cancels the other one. The arrow $a_1.\mathsf{try}(a_2, a_3)$ handles possible exception of a_1 by calling $[\![a_1]\!]$ with a progress value $P_i^1 :: p$ and an error handler continuation, which runs $\mathsf{adv}\ (P_i^2 :: \epsilon)$ to cancel a_1 and then runs a_3 if the handler is called. The arrow $\mathsf{noemit}(a)$ runs a with a progress value $Q :: p$ to suppress the progress made by arrows within a. Only when a itself completes, progress

can be made with $\mathsf{adv}\ p$ inside the continuation passed to $[\![a]\!]$. A recursive arrow with fix combinator is directly translated to a function with fix combinator. Lastly, an arrow a is run by passing unit value as input, empty progress value, and identity function as continuation and error handler.

2.3 Syntax of Translated Arrows

$e \in Exp$	$::=$	x	variable
	$\mid$	$\lambda x.e$	function
	$\mid$	$e\ e'$	call
	$\mid$	$e_1;\ e_2$	sequence
	$\mid$	$\mathsf{case}\ e\ \mathsf{of}\ \mathrm{succ}(y) \Rightarrow e_1,\ \mathrm{fail}(y) \Rightarrow e_2$	case expression
	$\mid$	$\mathsf{async}\ v_e\ e_p\ \lambda x.e$	async expression
	$\mid$	$\mathsf{adv}\ e_p$	advance the progress of arrow
	$\mid$	$\mathsf{fix}(\lambda\alpha.e)$	recursive arrow
	$\mid$	e_p	
e_p	$::=$	$p \mid P_i^j :: e_p \mid Q :: e_p \mid v_p$	progress expression
v_p	$::=$	$\epsilon \mid P_i^j :: v_p \mid Q :: v_p$	progress value
v_e	$::=$	$\mathsf{event}(f, v)$	event value

Figure 3: Syntax of translated arrows

The syntax of the translated arrow is summarized in Figure 3. The case expression is used for pattern-matching a value that corresponds to either the successful execution of an arrow or a failure caused by exception. The expression $\mathsf{async}\ v_e\ e_p\ \lambda z.e$ registers a pair of a progress expression e_p and a callback function $\lambda z.e$ to an event value v_e in the runtime environment. The expression $\mathsf{fix}(\lambda\alpha.e)$ defines a recursive function where α is a variable that recursively binds to the function itself. The expression $\mathsf{adv}\ e_p$ removes arrows using the progress expression e_p. A progress expression is a list of markers of the form of P_i^j or Q. The marker pair P_i^1 and P_i^2 is used to signal the progress of an arrow so that the arrows

associated with P_i^j is cancelled if an arrow associated with P_i^k makes progress, where $j \neq k$. The evaluation of $\mathsf{adv}\ e_p$ will proceed from left to right for each marker in e_p but it stops if the marker Q is reached. Effectively, the marker Q is used to stop the propagation of arrow cancellation.

2.4 Semantics of Translated Arrows

$$
\begin{array}{lcll}
\Delta, \mathcal{E}[\mathsf{async}\ v_e\ v_p\ \lambda z.e] & \rightarrow & \Delta[v_e \mapsto (v_p, \lambda z.e)], \mathcal{E}[()] & \text{R-Async} \\
\Delta, v & \rightarrow & \Delta \backslash \{v_e \mapsto (v_p, \lambda z.e)\}, [\mathbf{resp}(v_e)/z]e & \text{R-Event} \\
\Delta, \mathcal{E}[\mathsf{adv}\ P_i^j :: v_p] & \rightarrow & \Delta',\ \mathcal{E}[\mathsf{adv}\ v_p] & \text{R-Adv} \\
 & & \Delta' = \{v_e \mapsto (v_p', \lambda z.e) \in \Delta \mid P_i^k \notin v_p', k \neq j\} & \\
\Delta, \mathcal{E}[\mathsf{adv}\ Q_i :: v_p] & \rightarrow & \Delta, \mathcal{E}[()] & \text{R-Adv2} \\
\Delta, \mathcal{E}[\mathsf{adv}\ \epsilon] & \rightarrow & \Delta, \mathcal{E}[()] & \text{R-Adv3} \\
\Delta, \mathcal{E}[\mathsf{fix}(\lambda\alpha.e)] & \rightarrow & \Delta, \mathcal{E}[\ [\mathsf{fix}(\lambda\alpha.e)/\alpha]e\ ] & \text{R-Fix} \\
\Delta, \mathcal{E}[(\lambda x.e)\ v] & \rightarrow & \Delta, \mathcal{E}[\ [v/x]e\ ] & \text{R-Call} \\
\Delta, \mathcal{E}[v;\ e] & \rightarrow & \Delta, \mathcal{E}[e] & \text{R-Seq}
\end{array}
$$

Figure 4: Runtime semantics of translated arrows

The operational semantics of the translated arrows is shown in Figure 4 with reduction rules of program states represented by a context Δ and an expression e. Δ is a list of mapping from event value v_e to a pair of a progress object and a callback $(v_p, \lambda x.e)$. $\mathcal{E}$ is the evaluate context that represents a family of terms with a hole as defined below.

$$
\begin{array}{rcl}
\mathcal{E} & ::= & [.] \\
 & | & \mathcal{E}\ e \\
 & | & v\ \mathcal{E} \\
 & | & \mathcal{E};\ e \\
 & | & \mathsf{case}\ \mathcal{E}\ \mathsf{of}\ \mathrm{succ}(y) \Rightarrow e_1,\ \mathrm{fail}(y) \Rightarrow e_2
\end{array}
$$

By Rule R-Async, $\mathsf{async}\ v_e\ v_p\ \lambda x.e$ adds to Δ the mapping from v_e to a pair of a progress value and a callback. After the program state evaluates to a value Δ, v, events may be

handled. By Rule R-Event, if an event v_e fires, then its response $\mathbf{resp}(v_e)$ is passed to the callback registered to v_e and the mapping for this event is removed from the context Δ. By Rule R-Adv, the evaluation of $\mathsf{adv}\ P_i^j :: v_p$ removes any event callback paired with a progress value that contains P_i^k from Δ where $j \neq k$ and then evaluates $\mathsf{adv}\ v_p$. Rules R-Adv2 and R-Adv3 stop the evaluation of $\mathsf{adv}\ v_p$ when v_p is empty or marker Q is reached.

3 Parallel Asynchronous Arrows Detection

In this section, we present a set of formal rules for obtaining a list of the possibly-parallel arrows within a given construct, and provide some example applications of the rules provided. A user can use a utility function provided with the framework to analyze their program, in part or in full.

3.1 Analysis Rules

We present a set of functions for analyzing an arrow program to identify parallel arrows. The function $P(a, \sigma)$ is used to analyze an arrow a with context σ and yields a set of pairs of possibly-parallel arrows. The context is not used for most rules, but the recursive nature of the fix combinator requires its use, so we must carry it along as a result.

This is a 'may' analysis in that if we list two asynchronous arrows as potentially parallel, they *may be* parallel, but we cannot tell for sure; conversely, if we *do not* list a pair of arrows within a construct as potentially parallel, they *must not be* parallel (within that construct).

When looking for pairs of parallel, it is sometimes necessary to know all of the asynchronous arrows a given construct contains, for which we provide the function $A(a, \sigma)$. Likewise, in some cases we need to know only the first asynchronous arrow in a construct, for which we provide the function $F(a, \sigma)$. This is required in situations where the first asynchronous arrow to complete makes 'progress' and causes the cancellation of other 'competing' arrows, as in the case of an any combinator. So the analysis of the P function will often in-

volve the invocation of the A and F functions to provide the necessary information, and these functions in turn are often called recursively on a proper subset of the current arguments (thus we can guarantee that analysis will terminate).

At the end of the analysis, the user is presented with a set of pairs of arrows which may be parallel to each other. They can then analyze this set to check if it matches their expectation of their program. For example, if it contains asynchronous arrows that are indicated to be parallel to themselves, they may desire to investigate that arrow to ensure that it is safe for it to execute in that manner.

3.1.1 All Asynchronous Arrows

The first function we present is used to identify all of the asynchronous arrows within a given construct. This is necessary for constructs like the 'all' combinator, where all asynchronous arrows in each branch may be parallel to all others. The function A identifies all asynchronous arrows within a given construct:

$$
\begin{array}{lcl}
A(\mathsf{lift}(f), \sigma) & = & \emptyset \\
A(\mathsf{alift}(f), \sigma) & = & \{\mathsf{alift}(f)\} \\
A(a_1.\mathsf{seq}(a_2), \sigma) & = & A(a_1, \sigma) \cup A(a_2, \sigma) \\
A(a_1.\mathsf{on}(s, a_2), \sigma) & = & A(a_1, \sigma) \cup A(a_2, \sigma) \\
A(a_1.\mathsf{any}(a_2), \sigma) & = & A(a_1, \sigma) \cup A(a_2, \sigma) \\
A(a_1.\mathsf{all}(a_2), \sigma) & = & A(a_1, \sigma) \cup A(a_2, \sigma) \\
A(\mathsf{noemit}(a), \sigma) & = & A(a, \sigma) \\
A(a_1.\mathsf{try}(a_2, a_3), \sigma) & = & A(a_1, \sigma) \cup A(a_2, \sigma) \cup A(a_3, \sigma) \\
A(\mathsf{fix}(\lambda\alpha.a), \sigma) & = & A(a, \sigma) \\
A(\alpha, \sigma) & = & \begin{cases} \sigma(\alpha)[0] & \text{if } \alpha \in \mathrm{dom}(\sigma) \\ \emptyset & \text{otherwise} \end{cases} \\
A(\mathrm{boxed}(a), \sigma) & = & \{a\}
\end{array}
$$

Figure 5: Concurrency Detection: All Arrows

Most rules here are straightforward. It is clear, for example, that synchronous arrows (lift) are not asynchronous arrows, and asynchronous arrows (alift) are by definition asynchronous.

Analysis of the seq, any, all, and try combinators should clearly return any asynchronous arrows in any of their parts.

The noemit combinator prevents asynchronous arrows contained therein from reporting progress and potentially cancelling 'competing' arrows, but still may contain asynchronous arrows, all of which must be identified.

The context σ in all of the rules and in particular the $A(\alpha, \sigma)$ will be explained in more detail later, but for now it should be evident that if a term α is not in the context, it can have no asynchronous arrows. The fix combinator, described in some detail in the introduction, will use the context in later analysis, but for identification of all asynchronous arrows within, we simply drill further into the term.

The 'boxed' construct allows a user to 'wrap' a given set of arrows to prevent the analysis of that set and considers the whole thing as one asynchronous arrows (which can still be parallel to others). This can be particularly useful if we are not concerned about parallelism in that particular section, or have already analyzed it and determined it is not involved in the particular issue at hand.

Note that for the 'on' combinator, which itself *introduces* asynchrony between the registration and emission of the named event s on the specified emitter a_1, we ignore the asynchronous event registration and emission for the purposes of parallel arrow detection. There are several reasons for this simplification. One, the information necessary to make such a construct useful for parallel detection makes it quite cumbersome to encode, analyze, and present coherently. Two, the high usage of events for control flow would generate a huge number of results which would quickly become overwhelming. Last, it is not our goal to handle errors involving named events intermixed with other asynchrony.

3.1.2 First Asynchronous Arrows

Next, we identify the *first* arrow in a given construct, defining the function F. This is mostly useful with the any combinator, which races two or more arrows against each other; when one

arrow makes progress, the competing arrows are canceled, but their 'first' arrows may have already partially executed, so we must consider them as potentially parallel to the others.

$$
\begin{array}{lcl}
F(\mathsf{lift}(f), \sigma) & = & \emptyset \\
F(\mathsf{alift}(f), \sigma) & = & \{\mathsf{alift}(f)\} \\
F(a_1.\mathsf{seq}(a_2), \sigma) & = & \begin{cases} F(a_1, \sigma) \text{ if } F(a_1, \sigma) \neq \emptyset \\ F(a_2, \sigma) \text{ otherwise} \end{cases} \\
F(a_1.\mathsf{on}(s, a_2), \sigma) & = & F(a_1, \sigma) \\
F(a_1.\mathsf{any}(a_2), \sigma) & = & F(a_1, \sigma) \cup F(a_2, \sigma) \\
F(a_1.\mathsf{all}(a_2), \sigma) & = & F(a_1, \sigma) \cup F(a_2, \sigma) \\
F(\mathsf{noemit}(a), \sigma) & = & A(a, \sigma) \\
F(a_1.\mathsf{try}(a_2, a_3), \sigma) & = & \begin{cases} F(a_1, \sigma) \cup F(a_3, \sigma) & \text{if } F(a_1, \sigma) \neq \emptyset \\ F(a_2, \sigma) \cup F(a_3, \sigma) & \text{otherwise} \end{cases} \\
F(\mathsf{fix}(\lambda\alpha.a), \sigma) & = & F(a, \sigma) \\
F(\alpha, \sigma) & = & \begin{cases} \sigma(\alpha)[1] & \text{if } \alpha \in \mathrm{dom}(\sigma) \\ \emptyset & \text{otherwise} \end{cases} \\
F(\mathrm{boxed}(a), \sigma) & = & \{a\}
\end{array}
$$

Figure 6: Concurrency Detection: First Arrows

This function has a few twists, but should otherwise be mostly straightforward. The first arrow for a lone asynchronous arrow is itself, and a synchronous arrow has no first asynchronous arrow. The first arrow in a fix combinator will be the first arrow in its wrapped term. The α rule is similar to that of the A function; if a term α is not in the context, it can have no asynchronous arrows, and in particular no 'first' asynchronous arrow.

A seq combinator will hit its first asynchronous point after its first asynchronous arrow, so if the first arrow is synchronous, we look to the next, and so forth through any chained sequences as well. The any and all combinators will begin execution of each of their branches, so the 'first' arrow must include the 'first' arrow from each branch.

The on combinator will actually make progress upon receipt of the event, so we only need to consider the event source for potential parallelism, as we have mentioned previously that we will ignore the asynchronous events themselves for the purposes of this analysis.

The noemit combinator is interesting in this case as it wraps all internal arrows and

prevents them from emitting progress events, so it will not make progress until the end of the combinator. For this reason, we must consider *all* arrows inside to be part of the 'first' asynchronous arrow for that construct.

The try combinator will attempt to execute arrow a_1 and will follow with arrow a_2 on success or arrow a_3 on failure. If arrow a_1 has asynchronous component arrows, the first thereof would be the obvious candidate for the 'first' arrow, but if there are synchronous arrows preceding it and one of those throws an exception, the first asynchronous arrow to make progress will actually be the first asynchronous arrow in arrow a_3. If arrow a_1 has no asynchronous components, arrow a_1 will either succeed and pass to arrow a_2 or fail and fall to arrow a_3; in either case the latter component is our only chance for an asynchronous arrow. Note that if there is an exception in arrow a_2, it will not be handled by arrow a_3; to handle a failure in arrow a_2, you would need to wrap the whole try in another try.

And again, the boxed construct allows us to wrap a section of arrows and consider it all to be one asynchronous arrow for parallel detection purposes.

3.1.3 Parallel Asynchronous Arrows

And finally, we identify any *pairs of arrows* that may be parallel within a given construct, using the function P:

$$
\begin{array}{lcl}
P(\mathsf{lift}(f), \sigma) & = & \emptyset \\
P(\mathsf{alift}(f), \sigma) & = & \emptyset \\
P(a_1.\mathsf{seq}(a_2), \sigma) & = & P(a_1, \sigma) \cup P(a_2, \sigma) \\
P(a_1.\mathsf{on}(s, a_2), \sigma) & = & P(a_1, \sigma) \cup P(a_2, \sigma) \\
P(a_1.\mathsf{any}(a_2), \sigma) & = & P(a_1, \sigma) \cup P(a_2, \sigma) \cup \left\{ (a, a') \middle| \begin{array}{l} (a \in A(a_1, \sigma) \wedge a' \in F(a_2, \sigma)) \vee \\ (a \in F(a_1, \sigma) \wedge a' \in A(a_2, \sigma)) \end{array} \right\} \\
P(a_1.\mathsf{all}(a_2), \sigma) & = & P(a_1, \sigma) \cup P(a_2, \sigma) \cup \{(a, a') \mid a \in A(a_1, \sigma) \wedge a' \in A(a_2, \sigma)\} \\
P(\mathsf{noemit}(a), \sigma) & = & P(a, \sigma) \\
P(a_1.\mathsf{try}(a_2, a_3), \sigma) & = & P(a_1, \sigma) \cup P(a_2, \sigma) \cup P(a_3, \sigma) \cup \left\{ (a, a') \middle| \begin{array}{l} a \in A(a_1, \sigma) \wedge \\ a' \in A(a_3, \sigma) \end{array} \right\} \\
P(\mathsf{fix}(\lambda\alpha.a), \sigma) & = & P(a, \sigma[\alpha \mapsto [A(a, \sigma), F(a, \sigma)]]) \\
P(\alpha, \sigma) & = & \emptyset \\
P(\mathrm{boxed}(a), \sigma) & = & \emptyset
\end{array}
$$

Figure 7: Concurrency Detection: Parallel Arrows

Clearly within a single lift or alift there are no parallel arrows, and clearly seq and noemit consist of simply the parallel arrows within their components. For the boxed construct, we treat all internals as a single asynchronous arrow, so like alift, we report no internal parallel arrows.

The on combinator has a similar restriction here as before, keeping in mind that we ignore the asynchrony of the event itself, as discussed previously. This leaves simply the internal parallel arrows of its two component parts.

For the any, all, and try combinators, clearly we must include any parallel arrows within any of their components, but in each case there is a bit more to consider.

The all combinator will fully execute each branch, with no particular synchronization between them. Thus, we simply consider all asynchronous arrows in each branch to be potentially parallel to all asynchronous arrows from each other branch.

In the any combinator, each branch will begin execution, and the first to make progress will be allowed to continue on while the others are canceled. Thus, one branch will execute all of its arrows and the others will execute only up through their first asynchronous arrow.

One might think we could then just match up the 'first' arrows from each branch, but it turns out to be a little more complicated than that. The first arrow in competing branches cannot necessarily be stopped immediately (e.g., consider a request to a remote server), so it still has the ability to effect change before detecting that it has been canceled, and this change may happen when the 'winning' arrow is already well past its first asynchronous point. Thus, we have to consider the set of arrows where all asynchronous arrows of each branch may be parallel to the 'first' of any other branches.

The try combinator involves a similar technicality of progress and cancellation. In this case, we are particularly concerned about the scenario where an arrow within a_1 throws an exception while other asynchronous arrows are executing. For example, consider $a_1.\mathsf{try}(a_2, a_3)$ and $a_1 = a_x.\mathsf{all}(a_y)$, in which all arrows are asynchronous. The all combinator will kick off execution of both the a_x and a_y arrows. If an exception occurs in arrow a_x while arrow a_y is in-flight, the handler arrow a_3 may begin executing before arrow a_y is fully terminated. Thus, in the rule we must consider all asynchronous arrows in a_1 to be potentially parallel to all asynchronous arrows in a_3. We need not consider any interaction between arrows in arrows a_1 and a_2 as a_1 will have run to completion before execution of a_2 begins.

The fix combinator relies on processing its 'All' and 'First' rules for its wrapped term in order to build up context for usage of its recursive variable.

3.2 Examples

In this section we present some example applications of the rules. In order to make it easier to follow along, we will often use underlining to indicate the rule or term being processed in the current step.

3.2.1 Simple Example

Consider a simple example application of function P to $a_1.\mathsf{any}(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\mathsf{seq}(a_4))$, in which arrows a_1–a_4 are asynchronous. Suppose arrow a_1 represents loading a previously-

processed image from a server, arrow a_2 represents loading the unprocessed image from disk, arrow a_3 represents processing the image (so together arrows a_2 and a_3 load and process the image), and arrow a_4 represents saving the locally-processed image back to the cache. By considering the example, we should expect arrow a_1 to be potentially parallel to all the others, even with the noemit. The full application of the rules follows:

$$
\begin{array}{ll}
\underline{P}(a_1.\underline{\mathsf{any}}(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\mathsf{seq}(a_4))) & \Rightarrow \\
\underline{P}(a_1) \cup \underline{P}(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\underline{\mathsf{seq}}(a_4)) \cup & \\
\left\{(a,a') \middle| \begin{array}{l} (a \in A(a_1) \wedge a' \in F(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\mathsf{seq}(a_4))) \vee \\ (a \in F(a_1) \wedge a' \in A(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\mathsf{seq}(a_4))) \end{array} \right\} & \Rightarrow \\
\emptyset \cup \underline{P}(\underline{\mathsf{noemit}}(a_2.\underline{\mathsf{seq}}(a_3))) \cup P(a_4) \cup & \\
\left\{(a,a') \middle| \begin{array}{l} (a \in A(a_1) \wedge a' \in F(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\mathsf{seq}(a_4))) \vee \\ (a \in F(a_1) \wedge a' \in A(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\mathsf{seq}(a_4))) \end{array} \right\} & \Rightarrow \\
\underline{P}(a_2) \cup \underline{P}(a_3) \cup \underline{P}(a_4) \cup & \\
\left\{(a,a') \middle| \begin{array}{l} (a \in A(a_1) \wedge a' \in F(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\mathsf{seq}(a_4))) \vee \\ (a \in F(a_1) \wedge a' \in A(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\mathsf{seq}(a_4))) \end{array} \right\} & \Rightarrow \\
\emptyset \cup \emptyset \cup \emptyset \cup & \\
\left\{(a,a') \middle| \begin{array}{l} (a \in \underline{A}(a_1) \wedge a' \in \underline{F}(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\underline{\mathsf{seq}}(a_4))) \vee \\ (a \in \underline{F}(a_1) \wedge a' \in \underline{A}(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\underline{\mathsf{seq}}(a_4))) \end{array} \right\} & \Rightarrow \\
\left\{(a,a') \middle| \begin{array}{l} (a \in \{a_1\} \wedge a' \in \underline{F}(\underline{\mathsf{noemit}}(a_2.\mathsf{seq}(a_3)))) \vee \\ (a \in \{a_1\} \wedge a' \in A(\mathsf{noemit}(a_2.\mathsf{seq}(a_3))) \cup \underline{A}(a_4)) \end{array} \right\} & \Rightarrow \\
\left\{(a,a') \middle| \begin{array}{l} (a \in \{a_1\} \wedge a' \in \underline{A}(\underline{\mathsf{noemit}}(a_2.\mathsf{seq}(a_3)))) \vee \\ (a \in \{a_1\} \wedge a' \in \underline{A}(\underline{\mathsf{noemit}}(a_2.\mathsf{seq}(a_3))) \cup \{a_4\}) \end{array} \right\} & \Rightarrow \\
\left\{(a,a') \middle| (a \in \{a_1\} \wedge a' \in \underline{A}(a_2.\underline{\mathsf{seq}}(a_3)) \cup \{a_4\}) \right\} & \Rightarrow \\
\{(a,a') | (a \in \{a_1\} \wedge a' \in \{a_2\} \cup \{a_3\} \cup \{a_4\})\} & \Rightarrow \\
(a_1,a_2),(a_1,a_3),(a_1,a_4) & .
\end{array}
$$

Figure 8: Parallel arrow analysis of $a_1.\mathsf{any}(\mathsf{noemit}(a_2.\mathsf{seq}(a_3)).\mathsf{seq}(a_4))$

Thus our expectation is fulfilled.

3.2.2 Simple Example with '.on'

Consider $a_1.\mathsf{seq}(a_2).\mathsf{any}(e_1.\mathsf{on}(s, a_3.\mathsf{seq}(a_4)))$, where a_1–a_4 are asynchronous arrows, e_1 is an emitter (e.g. a button), and s is an event name. Upon analysis, we expect arrow a_1 to be potentially parallel to both arrow a_3 and arrow a_4, as the any construct pairs the first arrow

from each part with each asynchronous arrow from the other, but e_1 is synchronous, which eliminates that set. We omit the context (σ) in each call since this example does not require any context.

$$
\begin{array}{ll}
\underline{P}(a_1.\mathsf{seq}(a_2).\underline{\mathsf{any}}(e_1.\mathsf{on}(s, a_3.\mathsf{seq}(a_4)))) & \Rightarrow \\
\underline{P}(a_1.\underline{\mathsf{seq}}(a_2)) \cup \underline{P}(e_1.\underline{\mathsf{on}}(s, a_3.\mathsf{seq}(a_4))) \cup & \\
\left\{(a,a') \middle| \begin{array}{l}(a \in A(a_1.\mathsf{seq}(a_2)) \wedge a' \in F(e_1.\mathsf{on}(s, a_3.\mathsf{seq}(a_4)))) \vee \\ (a \in F(a_1.\mathsf{seq}(a_2)) \wedge a' \in A(e_1.\mathsf{on}(s, a_3.\mathsf{seq}(a_4))))\end{array}\right\} & \Rightarrow \\
\underline{P}(a_1) \cup \underline{P}(a_2) \cup \underline{P}(e_1) \cup \underline{P}(a_3.\mathsf{seq}(a_4)) \cup & \\
\left\{(a,a') \middle| \begin{array}{l}(a \in A(a_1.\mathsf{seq}(a_2)) \wedge a' \in F(e_1.\mathsf{on}(s, a_3.\mathsf{seq}(a_4)))) \vee \\ (a \in F(a_1.\mathsf{seq}(a_2)) \wedge a' \in A(e_1.\mathsf{on}(s, a_3.\mathsf{seq}(a_4))))\end{array}\right\} & \Rightarrow \\
\underline{\emptyset \cup \emptyset \cup \emptyset \cup (\emptyset \cup \emptyset)} \cup & \\
\left\{(a,a') \middle| \begin{array}{l}(a \in \underline{A}(a_1.\underline{\mathsf{seq}}(a_2)) \wedge a' \in \underline{F}(e_1.\underline{\mathsf{on}}(s, a_3.\mathsf{seq}(a_4)))) \vee \\ (a \in \underline{F}(a_1.\underline{\mathsf{seq}}(a_2)) \wedge a' \in \underline{A}(e_1.\underline{\mathsf{on}}(s, a_3.\mathsf{seq}(a_4))))\end{array}\right\} & \Rightarrow \\
\left\{(a,a') \middle| \begin{array}{l}(a \in \underline{A}(a_1) \cup \underline{A}(a_2) \wedge a' \in \underline{F}(e_1)) \vee \\ (a \in \underline{F}(a_1) \wedge a' \in \underline{A}(e_1) \cup \underline{A}(a_3.\mathsf{seq}(a_4)))\end{array}\right\} & \Rightarrow \\
\left\{(a,a') \middle| \begin{array}{l}(a \in \{a_1\} \cup \{a_2\} \wedge a' \in \emptyset) \vee \\ (a \in \{a_1\} \wedge a' \in \{a_3, a_4\})\end{array}\right\} & \Rightarrow \\
\{(a_1, a_3), (a_1, a_4)\} & .
\end{array}
$$

Figure 9: Parallel arrow analysis of $a_1.\mathsf{seq}(a_2).\mathsf{any}(e_1.\mathsf{on}(s, a_3.\mathsf{seq}(a_4)))$

Thus, arrow a_1 may be parallel to both arrow a_3 and arrow a_4, but no other arrows may be, as expected.

3.2.3 Introduction Arrow Example, Broken Form

Here we analyze the arrows example from the Introduction. As mentioned there, we expect it to report `createOneUser` parallel to itself. Assume *button* is a synchronous emitter and *callback* is synchronous. We use α instead of a and $\mathsf{fix}(\lambda\alpha.x)$ instead of $\mathsf{fix}(a \Rightarrow x)$ here to match the rule syntax and make it a bit easier to follow.

$\mathsf{fix}(\lambda\alpha.button.\mathsf{on}(\text{`}click\text{'}, \alpha.\mathsf{any}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha))))$

$$
\begin{array}{ll}
\underline{P}(\underline{\mathsf{fix}}(\lambda\alpha.button.\mathsf{on}(\text{`}click\text{'}, \alpha.\mathsf{any}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha)))), \sigma) & \Rightarrow \\
P(button.\mathsf{on}(\text{`}click\text{'}, \alpha.\mathsf{any}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha))), \sigma[\alpha \mapsto [& \\
\underline{A}(button.\underline{\mathsf{on}}(\text{`}click\text{'}, \alpha.\mathsf{any}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha))), \sigma), & \\
\underline{F}(button.\underline{\mathsf{on}}(\text{`}click\text{'}, \alpha.\mathsf{any}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha))), \sigma)]]) & \Rightarrow \\
P(button.\mathsf{on}(\text{`}click\text{'}, \alpha.\mathsf{any}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha))), \sigma[\alpha \mapsto [& \\
\underline{A}(button, \sigma) \cup \underline{A}(\alpha.\underline{\mathsf{any}}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha)), \sigma), & \\
\underline{F}(button, \sigma)]]) & \Rightarrow \\
P(button.\mathsf{on}(\text{`}click\text{'}, \alpha.\mathsf{any}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha))), \sigma[\alpha \mapsto [& \\
\emptyset \cup \underline{A}(\alpha, \sigma) \cup \underline{A}(createOneUser.\underline{\mathsf{seq}}(callback).\underline{\mathsf{seq}}(\alpha), \sigma), \emptyset]]) & \Rightarrow \\
P(button.\mathsf{on}(\text{`}click\text{'}, \alpha.\mathsf{any}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha))), \sigma[\alpha \mapsto [& \\
\emptyset \cup \underline{A}(createOneUser, \sigma) \cup \underline{A}(callback, \sigma) \cup \underline{A}(\alpha, \sigma), \emptyset]]) & \Rightarrow \\
P(button.\mathsf{on}(\text{`}click\text{'}, \alpha.\mathsf{any}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha))), \sigma[\alpha \mapsto [& \\
\{createOneUser\} \cup \emptyset \cup \emptyset, \emptyset]]) & \Rightarrow \\
\text{let } \sigma' ::= \sigma[\alpha \mapsto [\{createOneUser\}, \emptyset]] & \\
\underline{P}(button.\underline{\mathsf{on}}(\text{`}click\text{'}, \alpha.\mathsf{any}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha))), \sigma') & \Rightarrow \\
\underline{P}(button, \sigma') \cup \underline{P}(\alpha.\underline{\mathsf{any}}(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha)), \sigma') & \Rightarrow \\
\emptyset \cup \underline{P}(\alpha, \sigma') \cup \underline{P}(createOneUser.\underline{\mathsf{seq}}(callback).\underline{\mathsf{seq}}(\alpha), \sigma') \cup & \\
\left\{ (a_1, a_2) \middle| \begin{array}{l} (a_1 \in \underline{A}(\alpha, \sigma') \wedge a_2 \in \underline{F}(createOneUser.\underline{\mathsf{seq}}(callback).\underline{\mathsf{seq}}(\alpha), \sigma')) \vee \\ (a_1 \in \underline{F}(\alpha, \sigma') \wedge a_2 \in A(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha), \sigma')) \end{array} \right\} & \Rightarrow \\
\emptyset \cup \underline{P}(createOneUser, \sigma') \cup \underline{P}(callback, \sigma') \cup \underline{P}(\alpha, \sigma') \cup & \\
\left\{ (a_1, a_2) \middle| \begin{array}{l} (a_1 \in \{createOneUser\} \wedge a_2 \in \underline{F}(createOneUser, \sigma')) \vee \\ (a_1 \in \emptyset \wedge a_2 \in A(createOneUser.\mathsf{seq}(callback).\mathsf{seq}(\alpha), \sigma')) \end{array} \right\} & \Rightarrow \\
\emptyset \cup \emptyset \cup \emptyset \cup \{(a_1, a_2) | (a_1 \in \{createOneUser\} \wedge a_2 \in \{createOneUser\})\} & \Rightarrow \\
(createOneUser, createOneUser) & .
\end{array}
$$

Figure 10: Parallel arrow analysis of Introduction Arrow Example, Broken Form

Thus, as expected, we see that `createOneUser` is potentially parallel to itself.

3.2.4 Introduction Arrow Example, Fixed Form

And now we will analyze the 'fixed' arrows example from the Introduction, which should report that `createOneUser` is no longer parallel to itself. Again we assume that *button* is a synchronous emitter and *callback* is synchronous. We use α instead of a and $\mathsf{fix}(\lambda\alpha.x)$ instead of $\mathsf{fix}(a \Rightarrow x)$ here to match the rule syntax and make it a bit easier to follow.

$\mathsf{fix}(\lambda\alpha.button.\mathsf{on}(\text{`}click\text{'}, createOneUser).\mathsf{seq}(callback).\mathsf{seq}(\alpha))$

$$
\begin{array}{ll}
\underline{P}(\underline{\mathsf{fix}}(\lambda\alpha.button.\mathsf{on}(\text{`}click\text{'}, createOneUser).\mathsf{seq}(callback).\mathsf{seq}(\alpha)), \sigma) & \Rightarrow \\
P(button.\mathsf{on}(\text{`}click\text{'}, createOneUser).\mathsf{seq}(callback).\mathsf{seq}(\alpha), \sigma[\alpha \mapsto [& \\
\underline{A}(button.\mathsf{on}(\text{`}click\text{'}, createOneUser).\underline{\mathsf{seq}}(callback).\underline{\mathsf{seq}}(\alpha), \sigma), & \\
\underline{F}(button.\mathsf{on}(\text{`}click\text{'}, createOneUser).\underline{\mathsf{seq}}(callback).\underline{\mathsf{seq}}(\alpha), \sigma)]]) & \Rightarrow \\
P(button.\mathsf{on}(\text{`}click\text{'}, createOneUser).\mathsf{seq}(callback).\mathsf{seq}(\alpha), \sigma[\alpha \mapsto [& \\
\underline{A}(button.\underline{\mathsf{on}}(\text{`}click\text{'}, createOneUser), \sigma) \cup \underline{A}(callback, \sigma) \cup \underline{A}(\alpha, \sigma), & \\
\underline{F}(button.\underline{\mathsf{on}}(\text{`}click\text{'}, createOneUser), \sigma)]]) & \Rightarrow \\
P(button.\mathsf{on}(\text{`}click\text{'}, createOneUser).\mathsf{seq}(callback).\mathsf{seq}(\alpha), & \\
\sigma[\alpha \mapsto [\underline{A}(button, \sigma) \cup \underline{A}(createOneUser, \sigma) \cup \emptyset \cup \emptyset, \underline{F}(button, \sigma)]]) & \Rightarrow \\
P(button.\mathsf{on}(\text{`}click\text{'}, createOneUser).\mathsf{seq}(callback).\mathsf{seq}(\alpha), & \\
\sigma[\alpha \mapsto [\emptyset \cup \{createOneUser\}, \emptyset]]) & \Rightarrow \\
\text{let}\sigma' ::= \sigma[\alpha \mapsto [\{createOneUser\}, \emptyset]] & \\
\underline{P}(button.\mathsf{on}(\text{`}click\text{'}, createOneUser).\underline{\mathsf{seq}}(callback).\underline{\mathsf{seq}}(\alpha), \sigma') & \Rightarrow \\
\underline{P}(button.\underline{\mathsf{on}}(\text{`}click\text{'}, createOneUser), \sigma') \cup \underline{P}(callback, \sigma') \cup \underline{P}(\alpha, \sigma') & \Rightarrow \\
\underline{P}(button, \sigma') \cup \underline{P}(createOneUser, \sigma') \cup \emptyset \cup \emptyset & \Rightarrow \\
\emptyset \cup \emptyset \Rightarrow \emptyset & .
\end{array}
$$

Figure 11: Parallel arrow analysis of Introduction Arrow Example, Fixed Form.

And thus, no arrows herein are parallel to themselves anymore.

3.2.5 Double 'Fix' Arrow

Consider the application of the parallel arrow detection rules to the double-'fix' arrow $\mathsf{fix}(\lambda\alpha.a.\mathsf{seq}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)))))$, where both a and b are asynchronous arrows:

$$
\begin{array}{ll}
\underline{P}(\underline{\mathsf{fix}}(\lambda\alpha.a.\mathsf{seq}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta))))),\sigma) & \Rightarrow \\
P(a.\mathsf{seq}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)))),\ \sigma[\alpha \mapsto [\underline{A}(a.\underline{\mathsf{seq}}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)))),\sigma),\underline{F}(a.\underline{\mathsf{seq}}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)))),\sigma)]]) & \Rightarrow \\
P(a.\mathsf{seq}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)))),\ \sigma[\alpha \mapsto [\underline{A}(a,\sigma) \cup \underline{A}(\underline{\mathsf{fix}}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta))),\sigma),\underline{F}(a,\sigma)]]) & \Rightarrow \\
P(a.\mathsf{seq}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)))),\sigma[\alpha \mapsto [a \cup \underline{A}(b.\underline{\mathsf{seq}}(\alpha.\mathsf{any}(\beta)),\sigma),a]]) & \Rightarrow \\
P(a.\mathsf{seq}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)))),\sigma[\alpha \mapsto [a \cup \underline{A}(b,\sigma) \cup \underline{A}(\alpha.\underline{\mathsf{any}}(\beta),\sigma),a]]) & \Rightarrow \\
P(a.\mathsf{seq}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)))),\sigma[\alpha \mapsto [a \cup b \cup \underline{A}(\alpha,\sigma) \cup \underline{A}(\beta,\sigma),a]]) & \Rightarrow \\
P(a.\mathsf{seq}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)))),\sigma[\alpha \mapsto [a \cup b \cup \underline{\emptyset \cup \emptyset},a]]) & \Rightarrow \\
\underline{P}(a.\underline{\mathsf{seq}}(\mathsf{fix}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)))),\sigma[\alpha \mapsto [a \cup b,a]]) & \Rightarrow \\
\text{let } \sigma' ::= \sigma[\alpha \mapsto [a \cup b,a]] & \\
\underline{P}(a,\sigma') \cup \underline{P}(\underline{\mathsf{fix}}(\lambda\beta.b.\mathsf{seq}(\alpha.\mathsf{any}(\beta))),\sigma') & \Rightarrow \\
\emptyset \cup P(b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)),\sigma'[\beta \mapsto [\underline{A}(b.\underline{\mathsf{seq}}(\alpha.\mathsf{any}(\beta)),\sigma'),\underline{F}(b.\underline{\mathsf{seq}}(\alpha.\mathsf{any}(\beta)),\sigma')]]) & \Rightarrow \\
P(b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)),\sigma'[\beta \mapsto [\underline{A}(b,\sigma') \cup \underline{A}(\alpha.\underline{\mathsf{any}}(\beta),\sigma'),\underline{F}(b,\sigma')]]) & \Rightarrow \\
P(b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)),\sigma'[\beta \mapsto [b \cup \underline{A}(\underline{\alpha},\sigma') \cup \underline{A}(\underline{\beta},\sigma'),b]]) & \Rightarrow \\
P(b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)),\sigma'[\beta \mapsto [\underline{b \cup \{a \cup b\} \cup (\emptyset)},b]]) & \Rightarrow \\
P(b.\mathsf{seq}(\alpha.\mathsf{any}(\beta)),\sigma'[\beta \mapsto [a \cup b,b]]) & \Rightarrow \\
\text{let } \sigma'' ::= \sigma'[\beta \mapsto [a \cup b,b]] = \sigma[\alpha \mapsto [a \cup b,a],\beta \mapsto [a \cup b,b]] & \\
\underline{P}(b.\underline{\mathsf{seq}}(\alpha.\mathsf{any}(\beta)),\sigma'') & \Rightarrow \\
\underline{P}(b,\sigma'') \cup P(\alpha.\mathsf{any}(\beta),\sigma'') & \Rightarrow \\
\emptyset \cup \underline{P}(\alpha.\underline{\mathsf{any}}(\beta),\sigma'') & \Rightarrow \\
\underline{P}(\underline{\alpha},\sigma'') \cup \underline{P}(\underline{\beta},\sigma'') \cup \left\{(a_1,a_2) \middle| \begin{array}{l}(a_1 \in \underline{A}(\underline{\alpha},\sigma'') \wedge a_2 \in \underline{F}(\underline{\beta},\sigma'')) \vee \\ (a_1 \in \underline{F}(\underline{\alpha},\sigma'') \wedge a_2 \in \underline{A}(\underline{\beta},\sigma''))\end{array}\right\} & \Rightarrow \\
\emptyset \cup \emptyset \cup \left\{(a_1,a_2) \middle| \begin{array}{l}(a_1 \in \{a \cup b\} \wedge a_2 \in \{b\}) \vee \\ (a_1 \in \{a\} \wedge a_2 \in \{a \cup b\})\end{array}\right\} & \Rightarrow \\
((a,b) \cup (b,b) \cup (a,a) \cup (a,b)) & \Rightarrow \\
(a,b) \cup (b,b) \cup (a,a) & .
\end{array}
$$

Figure 12: Parallel arrow analysis of double 'Fix'

And thus what we expected: arrow a is *potentially parallel* to itself and to arrow b, and arrow b is *potentially parallel* to itself and to arrow a.

4 Implementation

In this section, we discuss the implementation of the alift construct and some the work required to enable the use of the arrows framework within the Node.js environment. We then show some examples of applying parallel arrow analysis to concurrency errors in real-world Node.js projects. We select our examples mainly from NodeCB, a database of concurrency errors in real-world Node.js projects, compiled by Wang et al. [16]. We describe each problem, distill its raw JavaScript form to a minimal working example, convert it to arrows, show the parallel arrows detected, and present a fixed arrow form that addresses the concurrency issues.

4.1 Arrows for Node.js

The alift construct enables the arrows framework to be used in a variety of new ways, including with Node.js applications. A large amount of work was thus undertaken to enable the arrows framework to be used with Node.js applications while maintaining compatibility with the existing browser/DOM usage.

This involved stitching the code back together where it was previously artificially separated, adding stubs as appropriate for each environment, and exporting the appropriate types and functions for use in the `arrows` module. Through the process, a significant amount of unit testing was also added to ensure continued operation of the core mechanics and to improve maintainability and enable future work with reduced risk.

In order to implement the alift rule for Node.js applications, we first had to determine which callback formats would be handled by the arrows framework, since the framework must provide the callback as a hook back to itself so it can process the result and handle arrow logistics. The error-first callback pattern is widespread in Node.js, used by most asynchronous methods exposed by the Node.js core API[2] as well as many popular third-party packages. Under this pattern, each callback should be provided two parameters; the

[2] `https://nodejs.org/api/errors.html#errors_error_first_callbacks`

first parameter represents an error if one was encountered or is just `null` if all went well, in which case the second parameter holds the result.

The near-ubiquitous use of this pattern made it the easy choice for support within the arrows framework, and other callback forms should be translatable into this form without much effort. For instance, for an asynchronous function which only provides the value to the callback, a simple wrapper-callback can be provided which simply provides a `null` 'error':

```
var a = ((x, cb) => noErr(x, (res) => cb(null, res))).liftasyncerr();
```

Listing 5: Converting a non–error-first callback for use with `liftasyncerr`.

Note that the arrow in Listing 5 takes two parameters, but the asynchronous function only takes one input. The arrow itself will also only take this one input, `x`, which is provided by the preceding arrow; the callback parameter `cb` is actually supplied by the arrow framework. This is because the framework uses the callback to retake control, check return values, and handle arrow logistics.

Thus, using `liftasyncerr`, asynchronous functions in Node.js can be lifted into arrows, and can then be composed with other `lift`'ed synchronous functions and combinators to form full arrow programs.

4.1.1 Get-Port Node.js Module Translated to Arrows

What follows is a simple example of alift within Node.js, based on a real Node.js package, get-port[3], implemented with promises. It includes the (MIT-licensed) JavaScript/promises source[4] and a translation into arrows. This module is used to simply obtain an available TCP port.

```
// Original JavaScript/promises implementation:
const getPort = options => new Promise((resolve, reject) => {
  const server = net.createServer();

  server.unref();
  server.on('error', reject);
```

[3] https://www.npmjs.com/package/get-port
[4] https://github.com/sindresorhus/get-port/blob/master/index.js

```
  server.listen(options, () => {
    const port = server.address().port;
    server.close(() => {
      resolve(port);
    });
  });
});
```

Listing 6: Original get-port Node.js module implementation.

The package essentially attempts to open a Node.js server on the port specified or the next available port, and returns a `Promise` object, which the caller can then use to wait for resolution. If the asynchronous call to 'listen' on (open) the port is successful, Node.js will activate the callback, and the promise will be 'resolved', allowing the caller to move on with the port value. If the 'listen' call fails, Node.js will raise an 'error' event, which then causes the promise to be 'rejected', which can be handled appropriately by the caller.

In the translation to arrows in Listing 7, we leave the synchronous parts alone for the most part and wrap the targeted asynchronous components in alift constructs as required, and we allow access to the local `server` and `options` variables (the latter being defined elsewhere) rather than trying to marshall them around between the various arrows. Furthermore, we allow the 'error' event to throw an error, which will then be caught by the arrows framework and handled appropriately through the error continuation, either by the failure branch of a try construct or via the default error handler, which is to print the error message and exit.

The `remember` combinator used here simply 'remembers' the input to the preceding arrow and supplies it as the input to the following arrow, in this case preserving the `port` from the `saveSvrPort` arrow to pass it along as the `getPort` arrow result. '`{}`' is provided as an unnamed positional argument to `closeAsync` as `saveSvrPort` sends a value the close doesn't actually need (which is then 'remembered' to be returned from the arrow, as discussed).

```
// translation to arrows/liftasyncerr:
const arrows = require('arrows')

// synchronous parts
const server = net.createServer();
server.unref();
server.on('error', err => {throw err});
```

```
// Arrows
const listenAsync = ((cb) => server.listen(options, cb)).liftasyncerr();
const saveSvrPort = (() => server.address().port).lift();
const closeAsync = (({}, cb) => server.close(cb)).liftasyncerr();

var getPort = listenAsync.seq(saveSvrPort).seq(closeAsync).remember();
```

Listing 7: Arrows implementation of get-port Node.js module.

Thus it is now possible to implement Node.js applications in the arrows framework, and thereby gain access to the additional program analysis and features provided by the framework.

4.2 Repeated Remote Requests

In many cases, repeated remote requests, e.g. a user clicking the same button to load an image several times in quick succession, causes concurrency errors in event handlers and other sections of code that the developer expected to run independently. We present two such examples.

4.2.1 NodeCB BugID 37: Long CreateUser Process and Impatient User

This is the same example from Section 1, explained in more detail.

NodeCB Bug ID:	37
Project Name:	TryGhost/Ghost
Bug report URL:	`https://github.com/TryGhost/Ghost/issues/1834`
Bug report Title:	Race condition when creating initial user could allow multiple users to be created.

Here is the distilled callback form from the Introduction with both the 'broken' and the 'fixed' versions:

```
// callback syntax - 'broken'
button.on('click', () => createOneUser(callback));

// callback syntax - 'fixed'
let active = false;
button.on('click', () => {
  if (!active) {
    active = true;
```

```
    createOneUser(() => {active = false; callback()});
  }
});
```

Listing 8: `createOneUser` callback implementation, revisited.

In this case, it would be obvious based on the bug report that there is a problem with the user creation, so you would know where to start looking. But it may still not be obvious from looking at this code in-situ what enables the bad behavior.

Our addition, within the arrows framework, allows us to explicitly indicate that the `createUser` arrow may be parallel to *itself*, which could clearly be a problem. As discussed in the rule application of Section 3.2.3, the analysis specifically indicates that *only* the pair $(createOneUser, createOneUser)$ herein may be parallel.

```
// Arrow Syntax
var x = Arrow.fix(a => button.on('click',
            a.any(createOneUser.seq(callback).seq(a))
        )
); x.run();
```

Listing 9: `createOneUser` broken arrow implmementation, revisited.

With the following fix, the analysis of this construct indicates no arrows may be in parallel any more, as per the full application of the rules in Section 3.2.4:

```
var x = Arrow.fix(a => button.on('click', createOneUser)
                       .seq(callback).seq(a)
); x.run();
```

Listing 10: `createOneUser` fixed arrow implmementation, revisited.

4.2.2 NodeCB BugID 1: Database Atomicity Violation

NodeCB Bug ID:	1
Project Name:	porygonco/porybox
Bug report URL:	`https://github.com/porygonco/porybox/issues/157`
Bug report Title:	Race conditions

This example involves adding ids to user entries in a database. In the following listing, the original code and identified resolution are provided, as discussed in the paper by Wang et al. [16] (the deleted code is proceeded with '`-`' while new code is proceeded with '`+`'). If the

function `addIdToArray` is called twice in quick succession with the same `ownerName`, e.g. in response to user input, the `findOne` function will asynchronously return a copy of the same user object. Each invocation will then add its own id to its copy, and save its copy back to the database. Whichever save runs last will override and trample the earlier save, losing that information. The fix is to atomically update the user in the database, ideally utilizing facilities provided by the database implementation.

```
// Original: Porybox#157: Atomicity violation caused by non-
//                        deterministic event triggering:
  function addIdToArray (ownerName, id) {
-   return User.findOne({name: ownerName}).then(user => {
-     user._ids.push(id);
-     return user.save();
-   });
+   db.user.update({name:ownerName},{$push:{_ids: id}})
  }
```

Listing 11: Porybox `addIdToArray` implementation.

If this example were implemented in arrows, it would look something like this:

```
// Find a user
const findOne = ((name, cb) => {db.findOne(name, cb);}).liftasyncerr();
// Add an id to a user and save to the database
const pushIdAndSave = ((user, cb) => { user.ids.push(nextId++);
                       db.save(user, cb); }).liftasyncerr();
// a sample user name, lifted to an arrow
const ownerName = "Fred".lift();

// update twice in quick succession - produces a race condition
// where only the last id is preserved
     ownerName.seq(findOne, pushIdAndSave)
.all(ownerName.seq(findOne, pushIdAndSave))
.run();
// The db entry only contains {2} now, instead of {1,2} - oops!
```

Listing 12: Arrows-based asynchronous database update.

This arrows implementation has the same problem as the original JavaScript implementation: two runs of the same (or similar) arrow are interleaved, and whichever `save` runs last will trump the previous `save`. When run on this example, the parallel detection algorithm indicates that `findOne` and `pushIdAndSave` are potentially run in parallel to themselves and to each other (technically, it will indicate that the full `liftasyncerr` parts of each may be parallel, but it is simpler to consider the representative names here). This, in particular the

`db.save` calls in the latter function, is a problem, and it should be clear to the user that some interleaving could be, and in fact is, happening here.

The solution is to use an asynchronous update provided by the database to find and update the user in one atomic process:

```
// 'atomically' add an id to a user
const updateUser = ((name, cb) => {
                      db.update(name, nextId++, cb)
                    }).liftasyncerr();
ownerName.seq(updateUser).all(ownerName.seq(updateUser)).run();
// Two entries added safely 'in parallel'
// The db entry now (properly) contains {1,2}
```

Listing 13: Arrows-based atomic database update.

In this case, the parallel detection algorithm will still indicate that `updateUser` may be parallel to itself, but on closer inspection we can confirm that the new parallelism is safe. If this were being analyzed as part of a larger program, it could now be wrapped in a boxed combinator to prevent the algorithm from looking inside.

4.3 Logical Atomicity Violations

There are quite a number of bugs in NodeCB labeled as 'Atomicity Violations' which involve the user taking some (unexpected) action which then creates an atomicity violation, like those covered in the previous section. There are also a few issues labeled as 'Atomicity Violations' where the violation is enabled by the logic of the implementation itself, not requiring any specific action from the user to trigger the error. We cover one of these issues in this section.

4.3.1 NodeCB BugID 57: Task Queue Management

NodeCB Bug ID:	57
Project Name:	OptimalBits/bull
Bug report URL:	`https://github.com/OptimalBits/bull/issues/370`
Bug report Title:	Jobs can be processed and left in the 'wait' state

Bull is described as 'the fastest, most reliable, Redis-based queue for Node' (Redis is 'an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker'). Essentially, Bull provides a task queue with several parallel or interleaved worker processes ('processors'). Some processors take tasks from the wait queue, activate, lock, and process them; others identify 'stalled' tasks—tasks with locks which have expired and not have been renewed by their host processors.

In this particular issue, one processor removed a task from the queue and attempted to asynchronously obtain a lock to process the task. While it waited patiently for the lock, another processor swapped in and noticed the task in the 'active' state *without a lock* (yet), considered it 'stalled', and sent it back to the 'wait' queue. Then the first processor successfully obtained the lock and processed the task, yet it remained in the 'wait' queue, eventually to be processed *again*.

```
function processorTask() {
  let task = mgr.getWaitingTask(); // take job from wait queue
  task.obtainLock(() => {          // lock - asynchronous!
    task.doStuff();                // process
    // schedule self to run again
    setTimeout(processorTask, processDelay);
  });
}

function cleanupTask() {
  // find an active+unlocked job & send it back to wait queue
  mgr.handleStalledTask(() => {
    // schedule self to run again
    setTimeout(cleanupTask, cleanupDelay);
  });
}
```

Listing 14: Bull task queue callback implementation.

If the full details of even a distillation of the example were provided, it quickly becomes obvious that the problem is not readily identifiable. In the version below written in arrows, it is still not clear where the problem exists. Parallel arrow analysis on this construct, however, will indicate the pair $(obtainLockArrow, handleStalledArrow)$ may be parallel. The `remember` combinator used here simply 'remembers' the input to the preceding arrow and supplies it as the input to the following arrow, in this case preserving the `task` to later

pass it along to the task runner (`doStuffArrow`). Its effects are synchronous, so it can be ignored for parallel arrow analysis. The `delay` combinator simply inserts a delay of the specified duration and then continues with the subsequent arrow.

```
const getTaskArrow = (() => mgr.getWaitingTask()).lift();
const obtainLockArrow =
   ((task,cb) => { task.obtainLock(cb); }).liftasyncerr().remember();
const doStuffArrow = ((task) => task.doStuff()).lift();
const handleStalledArrow =
  ((cb) => mgr.handleStalledTask(cb)).liftasyncerr();

const processTaskArrow = Arrow.fix(x => getTaskArrow
  .seq(obtainLockArrow).seq(doStuffArrow).delay(processDelay).seq(x));
const cleanupTaskArrow =
  Arrow.fix(x => handleStalledArrow.delay(cleanupDelay).seq(x));

processTaskArrow.all(cleanupTaskArrow);
```

Listing 15: Bull task queue arrows implementation.

In PR#379, the maintainers of Bull switched to a package called 'node-redlock' for their task-locking to "implement atomic locking with a guaranteed owner" (it was *not* just a 'simple' change). A similar strategy could be taken with the arrows example, which would eliminate the asynchrony of the `obtainLockArrow`, or at least replace it with a more general asynchronous `processTaskArrow`, which is safer to run in parallel with `handleStalledArrow`, with the lock race condition eliminated.

4.4 Both Branches

In this classification of concurrency errors, the program is expected to take one of two paths to accomplish a task, but in some special circumstances, the code takes actions from more than one of the paths. For instance, in the rule application example from Section 3.2.1, the request to the remote server raced with the local image processing; if the arrows were not configured properly, both branches could complete and cause some concurrency issues.

4.4.1 NodeCB BugId 17: getBlock

NodeCB Bug ID:	17
Project Name:	ipfs/js-ipfs
Bug report URL:	`https://github.com/ipfs/js-ipfs/issues/318`
Bug report Title:	Uncaught Error: no writecb in Transform class

In this database example, if a request is made to a database for a block of data that is not yet available, an event is registered to notify the caller when the data becomes available. If the data is already available in the cache, the notification is canceled and the data is returned.

```
function getBlock (key, cb) {
  const finish = (key, block) => { removeListener(key); cb(block); }
  addBlockListener(key, (block) => finish(key, block));
  db.has(key, (exists) => {
    if (!exists) return cb(); // event will fire when data is ready
    db.get(key, (block) => {
      finish(key, block);
    });
  });
}
```

Listing 16: The original `getBlock` callback implemenation.

A call to `getBlock` first registers for listeners for when the data block becomes available, and checks to see if data is already available. If data is available, it calls the `finish` function, which cancels the listener and invokes the callback `cb`. The corresponding data event only fires when data is actually *added* to the database, not if it is already available, so in most cases it should be safe to register, as either the data will be available and the registration will be immediately canceled (in the call to `finish`), or the function will exit quietly and allow the notification to provide the data at a later point.

The problem comes in when data is being asynchronously added to the database (with the `put` method) when the request for the corresponding block is started as well. This could be done in separate threads, but are placed here sequentially for simplicity:

```
db.put(key, data, () => {something});
getBlock(key, () => {something});
```

Listing 17: `getBlock` activates the callback twice.

If the `put` takes some time to process and a `getBlock` request comes in during that time, the `has` call is blocked by the database. The `put` then completes, and triggers the data event, which is then processed, calling `finish` and then the `cb`. Then the `has` is unblocked, calls `get`, and then `finish`, which then *also* calls the `cb`. The `cb` does not handle being called twice very well and in this case throws an exception.

An equivalent, though somewhat simplified, arrows implementation is provided here (the variable `key` is defined elsewhere and provides the database key of the data for which we are looking):

```
var getNewBlockArr = ((cb) => addBlockListener(key, (err, block) =>
      { finish(key, block); cb(err, block); })).liftasyncerr();

var hasBlkArr = ((cb) => db.has(key, cb)).liftasyncerr();

var getBlkArr = ((cb) => db.get(key, (err, block) =>
      { finish(key, block); cb(err, block); })).liftasyncerr();

var getBlockNowArrow = noemit(hasBlkArr.seq(getBlkArr));

var getBlock = getNewBlockArr.any(getBlockNowArrow);
```

Listing 18: `getBlock` arrow translation.

Arrow `getNewBlockArr` registers `finish` to run when the listener on the new block of data for `key` is triggered, and then calls the framework-provided callback (this is somewhat unnatural in this case, but is closer to the original implementation). Arrows `hasBlkArr` and `getBlkArr` both wrap their respective asynchronous functions, with `getBlkArr` also calling `finish`. The two are then combined into `getBlockNowArrow`, wrapped in a noemit combinator to ensure simple progress of `hasBlkArr` does not impede the `getNewBlockArr`. Finally, `getBlock` races the two composite arrows against each other.

Parallel arrows analysis of this implementation yields {(`getNewBlockArr`, `hasBlkArr`), (`getNewBlockArr`, `getBlkArr`)}, with the pair containing `getBlkArr` causing the concurrency issues here (if `getBlkNowArr` has proceeded to `getBlkArr`, it has already committed to getting the data and calling `finish`).

Initially the ipfs maintainers just protected the callback from being called multiple times

within the `finish` function, but that didn't really resolve the root cause of the issue. Eventually in later versions, through at least one major refactoring, they moved the notification registration into the 'else' section of the `has`, so it will only register for the notification if it doesn't have the data in the first place. This is similar to the following arrows implementation, using try:

```
var finishArr = finish.lift();
var getBlock = hasBlkArr.try(getBlkArr, getNewBlockArr).seq(finishArr);
```

Listing 19: Fixed `getBlock` arrow implementation.

Now the parallel arrows analysis yields {(`hasBlkArr`, `getNewBlockArr`)}, which makes sense; it will check if it `has` the data, and proceed to `get` the data if so. If the `has` fails, it will fall back to registering for the notification. The potential parallelism remaining is due to the exception handling path; if `has` throws an exception at any point in its execution, it will be handled by invoking the `getNewBlockArr`, as discussed in Section 3.1.3. So this parallelism is expected, and could be ignored by wrapping the construct in a boxed construct, if so desired.

5 Related work

5.1 On the Analysis of Bugs in JavaScript

Wang, et al. [16] present a categorized database of Node.js concurrency bugs and discuss potential resolution strategies for similar errors. They found that two thirds of the bugs analyzed were caused by atomicity violations. They describe why concurrency bugs in Node.js (or server-side JavaScript in general) are difficult to find and what sets them apart from similar bugs in other asynchronous programming systems. In their analysis, almost all of the bugs studied led to severe failures of the affected application. We borrow several examples from their database, implement them in the arrows framework, and show how parallel arrow analysis helps to identify concurrency errors.

Mutlu et al. [12] attempt to limit the scope of "races that matter" by focusing on bugs which cause problems in persistent state (e.g. cookies, `localStorage`, or server-side storage) or semi-persistent state (e.g. `sessionStorage`, cleared on browser restart). They argue that the browser runtime model is basically designed to quietly ignore unhandled errors in general, and users have been 'trained' over time to handle such transient or 'benign' bugs as well, often by simply reloading the page. Their work also requires an specially-instrumented browser and DOM, used to track data propagation through memory and storage operations. Our work, in contrast, does not venture to specifically differentiate between concurrency bugs or place any intrinsic value thereon, but also does not require an instrumented browser to perform our analysis.

Hanam et al. [6] research common bug and fix patterns in JavaScript using machine learning on bug fix commits in open source repositories. Their aim is to help tool developers target the most prevalent bugs and provide strategies for finding and fixing bugs. They look at three different ways of grouping defects: by fault or root cause, typically used by static analysis tools; by symptom/presentation, typically used by dynamic analysis tools; or by repair, typically identified by automated repair tools, and how their research can be used to improve each. Our work fits within the static analysis category, so it would target errors their tool could identify based on fault or root cause.

5.2 On Bug Detection in JavaScript

Zheng et al. [17] perform static analysis to identify possible data inconsistencies and atomicity violations involving asynchronous calls. Their analysis is limited specifically to client-side JavaScript code by processing application code to extract the client-side code for automatic analysis, and is thus restricted to client-side scripting. The JavaScript landscape has also changed fairly dramatically since their research was published, so it obviously does not account for developments in Node.js or promises. Our work supports newer developments in the JavaScript language including the emergence of server-side scripting and Node.js.

Petrov et al. [14] present WEBRACER, a dynamic race detection tool for web applications. It relies on an instrumented browser that helps track memory access and uses their happens-before relationship definition to identify possible race conditions, and they were successful in locating such race conditions in several Fortune 100 company websites. There are limitations to their process, and it does generate quite a few false positives, which follow-up work by Raychev et al. [15] seeks to mitigate. Our analysis is, once again, static, and does not require software specifically instrumented for the analysis.

Raychev et al. [15] discuss the prevalence of harmful race conditions and found many such errors, even in the online presence of many of the largest public and privately held companies in the United States. They present a dynamic race detection tool called EVENTRACER which improves upon existing race detection solutions by introducing the concept of 'race coverage', which helps significantly reduce the number of false-positives by ruling out races 'covered' by another detected race. They plan to improve on their work by utilizing static analysis techniques to positively identify synchronization variables and further reduce false-positives. However, their solution is still dynamic and requires appropriate instrumentation and coverage to detect races, and ours provides static analysis to help identify concurrency issues.

Hong et al. [7] presents WAVE, the "Web Application's Virtual Environment", a testing framework for client-side JavaScript applications. It records a 'monitored execution' and then essentially reorders interactions in various permutations to see if that brings to light any concurrency bugs. This is obviously very different from the arrow composition and static analysis we provide, but does provide an interesting approach to identifying runtime issues with the non-determinism we also seek to handle.

5.3 On JavaScript Semantics

Work by Madsen et al. [11] introduces the 'event-based call graph' for static analysis of event-based Node.js JavaScript programs, and shows how it can be used to find dead listeners and

dead emits as well as some information about mismatched synchronous/asynchronous calls and unreachable functions. Their work, however, requires modeling of any modules that use native C++, which includes all of the core Node.js framework components. They report that their analysis appears to be useful for small Node.js programs. Our work does not look into the specific function being lifted, so we do not need to model Node.js framework components in order to make use of them.

Madsen et al. [10] then discuss some of the benefits and pitfalls of using promises, introduced in ECMAScript 6. Essentially they argue that promises are substantially better than pure event-driven JavaScript, but still have a lot of problems, on which we agree. They identify at least seven different types of problems they found in promise-based code, and introduce the promise graph to help programmers reason about their promise-based code and identify some of the errors discussed in the paper. This work is tangential to our work in arrows, but shows that promises are not a perfect solution to the asynchronous problem in JavaScript.

Loring, et al. [9] investigate the semantics of JavaScript, and in particular the Node.js runtime, in an effort to support future work on static and dynamic analysis of applications or to evaluate improvements to the framework itself. They investigate the semantics of node event queues and identify the differences between the two main asynchronous scheduling mechanisms, `process.nextTick` and `setImmediate`.

Gallaba et al. [5] present an empirical study characterizing JavaScript callback usage. They study the error-first callback convention, the Async.js library[5], and promises, and argue that program analysis of JavaScript must take into account asynchrony and its effects on control and information flow. Simplifying assumptions about callbacks and asynchrony in general might invalidate such analysis or lead to invalid results. Subsequent work [4] presents a tool called PROMISESLAND for refactoring JavaScript callbacks, including those utilizing the error-first convention, into JavaScript promises.

[5] `http://caolan.github.io/async/`

They also discuss related tools and frameworks used to help manage asynchrony in-place, including one in particular called bluebird [1], a promise library for JavaScript which seeks to improve debugging for JavaScript promises, including support for 'long stack traces.' Many such libraries have mechanisms for wrapping existing asynchronous libraries into 'promisified' code that can more easily be integrated into other promise-based code. However, even bluebird still cannot statically analyze possible parallelization like we can do within the arrows framework. There is currently no direct translation available for converting a JavaScript program written with callbacks or promises into the arrows framework, which could represent and area of further research and future work.

The work of Fritz et al. [3, 2] built upon work by Khoo et al. [8], which proposes arrows as a generalization of *monads* to help coordinate execution of JavaScript applications. Arrows serve similar purposes as promises in JavaScript to make callback logic easier to follow as sequential logic, but arrows provide static capability in analyzing JavaScript programs as we have demonstrated in this paper.

6 Conclusion and Future Work

In this work, we have shown that concurrency errors are a problem in JavaScript, and while newer language features like promises and libraries like bluebird help to alleviate some of the problems associated with raw callbacks, promises themselves are not without their own difficulties. We've presented the arrows framework, and in particular our parallel arrow analysis, as an alternative to promises, and have shown how it can help to locate concurrency errors.

Future work could be undertaken to improve the usability and power of the parallel analysis within the arrows framework. For example, the arrows framework is currently limited to passing along the asynchronous result (in the callback) and does not provide access to any synchronous results from the asynchronous call. There are some asynchronous

functions which return meaningful synchronous data, and workflows using those functions could be better supported if the proper semantics are determined. Events could also be considered, perhaps in a configurable manner, for analysis with parallelism.

The syntax required for the arrows framework to trap the callback in some cases is fairly cumbersome, and forms the basis for some additional work. Consider the example in Section 4.2.2. While the `findOne` arrow appears to take two parameters, `name` and `cb`, only one is actually provided by the preceding arrow; the `cb` is supplied by the arrow framework. This syntax, or a similar *but even more awkward* syntax, is required because the framework must provide the callback parameter explicitly, as it must retake control to check return values and handle arrow logistics. Future work could investigate improvements to this syntax, or perhaps a replacement syntax.

One such alternative syntax, similar to that of Haskell arrows notation using the `proc`-expression, as described by Ross Patterson [13], is already in active development, which could also simplify some of the mechanics of marshalling data between arrows.

References

[1] Bluebird. Features—Bluebird. `http://bluebirdjs.com/docs/features.html`, Aug 2016.

[2] Eric Fritz and Tian Zhao. Inferring types for asynchronous arrows in JavaScript. Technical report, Technical report, University of Wisconsin-Milwaukee, 2015.

[3] Eric Fritz and Tian Zhao. Typing and semantics of asynchronous arrows in JavaScript. Science of Computer Programming, 141:1–39, 2017.

[4] Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. Refactoring asynchrony in JavaScript. In Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, pages 353–363. IEEE, 2017.

[5] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. Don't call us, we'll call you: Characterizing callbacks in JavaScript. In Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on, pages 1–10. IEEE, 2015.

[6] Quinn Hanam, Fernando S de M Brito, and Ali Mesbah. Discovering bug patterns in JavaScript. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 144–156. ACM, 2016.

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

[8] Yit Phang Khoo, Michael Hicks, Jeffrey S Foster, and Vibha Sazawal. Directing JavaScript with arrows. In ACM Sigplan Notices, volume 44, pages 49–58. ACM, 2009.

[9] Matthew C Loring, Mark Marron, and Daan Leijen. Semantics of asynchronous JavaScript. In ACM SIGPLAN Notices, volume 52, pages 51–62. ACM, 2017.

[10] 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.

[11] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven Node.js JavaScript applications. In ACM SIGPLAN Notices, volume 50, pages 505–519. ACM, 2015.

[12] 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.

[13] Ross Paterson. A new notation for arrows. ACM SIGPLAN Notices, 36(10):229–240, 2001.

[14] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In ACM SIGPLAN Notices, volume 47, pages 251–262. ACM, 2012.

[15] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In ACM SIGPLAN Notices, volume 48, pages 151–166. ACM, 2013.

[16] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. A comprehensive study on real world concurrency bugs in Node.js. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pages 520–531. IEEE Press, 2017.

[17] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically locating web application bugs caused by asynchronous calls. In Proceedings of the 20th International Conference on World Wide Web, pages 805–814. ACM, 2011.