Type Inference of Asynchronous Arrows in JavaScript

Eric Fritz Tian Zhao
University of Wisconsin-Milwaukee
{fritz,tzhao}@uwm.edu

Abstract

Asynchronous programming with callbacks in JavaScript leads to code that is difficult to understand and maintain. Arrows, a generalization of monads, are an elegant solution to asynchronous program composition. Unfortunately, improper arrow composition can cause mysterious failures with subtle sources. We present an arrows-based DSL in JavaScript which encodes semantics similar to ES6 Promises and an optional type-checker that reports errors at arrow composition time.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—Software libraries; D.2.5 [*Software Engineering*]: Testing and Debugging—Error handling and recovery

Keywords Type Systems, Type Inference, Arrows, JavaScript, Asynchronous Programming, Concurrent Programming

1. Introduction

Event programming is prevalent in JavaScript. As the *lingua franca* of the web, it is responsible for driving a huge amount of user-interactive web applications. Because JavaScript is commonly executed in a single thread, blocking or long-running computations can often cause the page or entire browser to appear unresponsive. As a result, JavaScript programs are written in an event-driven style where programs register callback functions with the event loop. A callback function is dispatched by the event loop when an external event occurs, and control returns to the event loop once a callback function completes execution.

Heavy use of callbacks make control flow difficult to trace. Application logic becomes intimately mixed with sequencing logic. A single unit of application code may no longer be confined to one easily-readable function, but split arbitrarily far across a number of functions. This greatly decreases code understandability and maintainability.

The introduction of Promises in ES6 demonstrates a desire to reduce the complexity of callback-driven programs. Promises allow callbacks to be chained instead of nested, regaining some imperative flow of control. Similarly, Arrowlets [6] has demonstrated an elegant solution to composing callback functions by wrapping them in opaque units of execution using continuation functions. These units of execution encode *arrows* [5], which is a generalization of monads [11].

However, JavaScript lacks properties that make function, arrow, or promise composition palatable. Illegal compositions are not forbidden at composition time and often crash or lead to subtle behavioral issues at runtime. Frustratingly, the source location which displays incorrect behavior is often completely independent of the source location of the actual error, making the associated stack trace less than helpful. These errors force the developer to trace the arrow execution path backwards from the source of a runtime error, continuation-function by continuation-function, until the erroneous composition presents itself. Despite the benefits, this seems to leave the developer no better off than using callbacks during debugging.

Fortunately, there is a clear separation between the composition time and the execution time of arrows. It is possible to detect errors after the arrows have been composed but before their actual execution starts. To this end, we have developed an optional type-checker which infers and attaches a type to every arrow at composition time describing its input and output constraints and forbids the composition of two arrows that are not *type-composable*. This reduces a rather large class of errors during composition related to input/output clashes and requires only that the user adds an annotation to functions which are lifted into arrows.

This type-checker runs in pure JavaScript at program runtime and thus requires no pre-processing step. While the type-checker does not find errors prior to runtime, it does find errors prior to *arrow execution-time*. This technique effectively moves the source of errors from the point where an error may be observed to the point where an erroneous composition occurs. We have found this relocation of error messages invaluable and feel that moving errors earlier in runtime (without moving them completely outside of runtime) still provides a great benefit. The type-checker may be disabled, returning the program to the original runtime semantics without dynamic type-checks.

Our Contributions The main contributions of this paper are

1. an encoding of arrows which handles asynchronous errors in a manner similar to ES6 Promises, and
2. and an optional type system to aid developers with type-directed composition of asynchronous arrows.

The remainder of this paper is organized as follows. Section 2 provides a motivating example. Section 3 introduces the arrow constructors and combinators in our library and discusses their runtime semantics and encoding. Section 4 provides details of the type inference system and presents typing rules. Section 5 discusses the runtime cost and the development overhead of our library. Section 6 presents related work and Section 7 concludes. Our arrows library, the type-checker, and some sample applications are freely available [1].

Submitted to the Workshop on Reactive and Event-based Languages & Systems, 2015.

[1] http://arrows.eric-fritz.com

```
const makeConf = (resource, id) => {
  'url'      : '/api/v1/' + resource + '/' + id,
  'dataType': 'json'
};

var ajaxA = new AjaxArrow(id => {
  /* @conf :: Number
     @resp :: {a: Number} */
  return makeConf('a', id);
});

var ajaxB = new AjaxArrow(id => {
  /* @conf :: Number
     @resp :: {b: Number} */
  return makeConf('b', id);
});

var ajaxC = new AjaxArrow(id => {
  /* @conf :: Number
     @resp :: {c: Number} */
  return makeConf('c', id);
});

function log() {
  console.log(arguments);
}

const getId = () => /* @arrow :: T ~> Number */ 3;
const onErr = ex => {
  /* @arrow :: AjaxError ~> () */
  console.log('Remote server problem.');
}

const getA = o => /* @arrow :: {a: `x} ~> `x */ o.a;
const getB = o => /* @arrow :: {b: `x} ~> `x */ o.b;
const getC = o => /* @arrow :: {c: `x} ~> `x */ o.c;

Arrow.catch(
  Arrow.seq(
    Arrow.lift(getId),
    ajaxA, Arrow.lift(getA),
    ajaxB, Arrow.lift(getB),
    ajaxC, Arrow.lift(getC),
    Arrow.lift(log)),
  Arrow.lift(onErr)
).run()
```

Figure 1. Arrow example

2. Example

To illustrate the utility of our type inference tool, consider the example in Figure 1. Three Ajax calls are made to retrieve data from a remote sever, and the resource path of each call depends on a result from the previous call. The final result is printed to the console, and any errors are caught and logged.

The details of the arrow methods are explained in Section 3. For now, it is sufficient to understand that `lift` converts a function into an arrow, `seq` chains two or more arrows in sequence, `catch` executes the first arrow and uses the second as an exception handler, and an arrow does not begin execution until its `run` method is called. Functions which are *lifted* into arrows are generally annotated with a type. If a function does not have an annotation we assume that the function can accept anything and may return anything.

Without type-checking, a number of runtime errors could occur from this example. Composition of arrows could be misaligned such that the output of one arrow does not conform the input of another. There are at least seven points of potential failures in this small example. As remote requests take a long time to complete, runtime debugging for these kinds of programs may be particularly frustrating.

Using our type inference tool, developers can add type annotations for functions which are transformed into arrows. This enables typing errors to be discovered as early as possible. In particular, illegal composition of arrows would be discovered at composition time before any Ajax request is attempted. For example, `getA` expects its input to have a record type $\{a : {}'x\}$, where ${}'x$ is a type variable. We ensure that the result type $\{a : Number\}$ of the preceding arrow `ajaxA` can be unified with this type.

The result of `ajaxA` is type-checked at arrow execution time to ensure that the result matches the annotated type. Even though this dynamic type-check takes place after composition time, it still moves a possible error from the point where an unexpected value is read to the point where it is created. Such runtime checks can be disabled after testing.

3. Arrows

An arrow is a composable, opaque unit of execution which, in this context, runs in an asynchronous manner. An arrow may receive a number of arguments, but may only receive them from another arrow. Similarly, an arrow may produce a value, but that value may only be consumed by another arrow.

We embed a typed domain-specific language based on arrow operations into JavaScript. The host language may `lift` a function into an arrow, `run` an arrow, or `cancel` a running arrow. Arrows are meant to replace operations in JavaScript which were primarily asynchronous or callback-driven. As a result, values cannot flow from arrows back into the host language.

Definition 3.1 (Async Point)**.** The point in the execution of an arrow which requires an external event to continue is called an *async point*. These events include timers (e.g. `setTimeout`), user events (e.g. *click*, *keydown*), network events (e.g. *Ajax* calls), and certain arrow-specific actions (discussed in Section 3.2). Concurrent execution of other arrows or host-language code may occur within a blocked arrow's async point.

Definition 3.2 (Asynchronicity)**.** We say an arrow is *asynchronous* if its execution contains at least one async point. A running arrow may be canceled only if it is asynchronous, and it may be canceled *only* at an async point. Canceling an arrow effectively unregisters all of its active event handlers so that it is never notified to resume execution when an external event occurs.

Definition 3.3 (Progress Event)**.** An arrow may emit a *progress event* if it successfully resumes execution after blocking at an async point. These events may be explicitly suppressed (discussed in Section 3.2).

An overview of the primitives of our library follows. The arrow primitives consist of constructors and combinators. Arrow constructors create simple arrows from composition-time values. These arrows can transform data synchronously and handle asynchronous events. Arrow combinators compose a set of arrows to form workflow that can be linear, parallel, or repeating. The design and implementation of the library is heavily inspired by both Arrowlets [6] and ES6 Promises. We have, however, made a few major interface changes which are discussed in Section 6.

3.1 Constructors

We provide seven arrow constructors, detailed below.

Ajax The ajax arrow, denoted `ajax`(c), produces a value by issuing a remote HTTP request. The request parameters (e.g. *url*, *method*, *headers*, *request body*) are returned by the host-language configuration function c. If type-checking is enabled, it is expected that c is annotated with the input constraints of c and the expected result from the remote server. Dynamic type-checks are inserted following a successful response from the remote server to ensure the shape of the data matches the annotated type.

```
var state = Arrow.ajax(zip => {
  /* @conf :: Number
     @resp :: { city: String, state: String } */
  return {
    url      : '/api/v2/zip_codes/US/' + zip,
    dataType: 'json'
  };
});
```

Delay The delay arrow, denoted `delay`$(duration)$, passes along its own input, unmodified, after *delay* milliseconds pass. This arrow is asynchronous.

Elem The element arrow, denoted `elem`$(selector)$, produces a jQuery object (or possibly empty set of objects) matching the given selector.

Event The event arrow, denoted `event`$(name)$, takes an element as input and produces a *name*-event value *once that event occurs on the given element*. This arrow is asynchronous.

Lift A lifted arrow, denoted `lift`(f), produces a value determined by $f(x)$, where x is the input of the arrow and f is a host-language function. If type-checking is enabled, it is expected that f is annotated with the input and output constraints of f. Dynamic type-checks are inserted following the invocation of f to ensure the return value matches the annotated type.

```
var strmul = Arrow.lift((s, n) => {
  /* @arrow :: (String, Number) ~> Number */
  var acc = '';
  for (var i = 0; i < n; i++) acc += s;
  return acc;
});
```

Nth The nth arrow, denoted `nth`(n), takes a tuple of *at least* n elements as input and extracts its nth element.

Split The split arrow, denoted `split`(n), takes a single value v as input and converts it to an n-tuple, where each element of the tuple is v. This arrow attempts to preclude aliasing by creating n clones of the value v. This avoids problems with mutable references to values held by concurrently executing arrows.

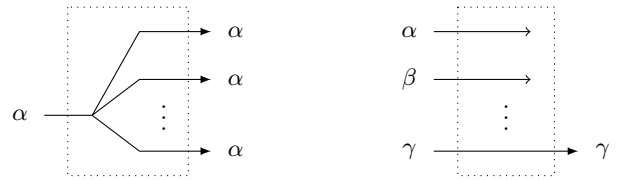

Figure 2. Dataflow diagrams for `split` and `nth` arrows.

Note that the `elem` constructor can be encoded by `lift`, but is provided for convenience. The `split` and `nth` constructors can also be encoded by `lift`, but their types depend on a compile-time value and cannot be annotated statically with an accurate type.

3.2 Combinators

We provide six arrow combinators, detailed below. The `repeat` and `noemit` combinators transform a single arrow, where the remaining five combinators transform a set of $n \geq 1$ arrows. Async points are represented in dataflow diagrams as double-slashed lines.

Seq The sequence combinator, denoted `seq`$(a_1, \ldots, a_n)$, composes n arrows which execute in order. The result of arrow a_i is fed into arrow a_{i+1}. The input to a_1 is the input of the combinator, and the result of the combinator is the result of a_n.

This combinator is asynchronous if *any* arrow a_i is asynchronous. The set of async points of the combinator is the union of the async points of each arrow a_i.

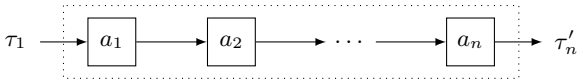

Figure 3. Dataflow diagram for the `seq` combinator.

This combinator generalizes the binary combinator

$$(a \ggg b) : (A \rightsquigarrow B) \to (B \rightsquigarrow C) \to (A \rightsquigarrow C)$$

in the arrow calculus [5].

Try The try combinator, denoted `try`(a, a_s, a_f), attempts to execute a with the input of the combinator. If no error occurs during the execution of a, its output is fed into the success arrow a_s. Otherwise, the error value is fed into the failure arrow a_f. The result of the combinator is either the result of arrow a_s or arrow a_f, depending on which one executed at runtime.

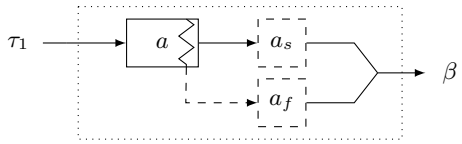

Figure 4. Dataflow diagram for the `try` combinator.

This combinator is *definitely* asynchronous if all control flow paths through the arrow contain an async point. This is guaranteed only when both arrow a_s and arrow a_f are asynchronous, as arrow a may halt with an error before its first async point.

Promise's `then` and `catch` methods can be encoded by the `try` combinator. The statement p.`then`$(resolve)$ executes p and then the callback *resolve* on successful execution. The statement p.`catch`$(reject)$ executes p and, if an error occurs, calls *reject* with the error as input. The statement p.`then`$(resolve,\ reject)$ executes p and then calls either the callback *resolve* or *reject* on successful or unsuccessful execution, respectively. The *reject* callback is not executed if an error occurs in *resolve*.

We can encode these statements with the `seq` combinator, the `try` combinator, and an *identity* arrow, id, as follows, where the arrow a is functionally equivalent to the promise p.

$$\begin{aligned} p.\texttt{then}(s) &\equiv \texttt{seq}(a,\ \texttt{lift}(s)) \\ p.\texttt{catch}(f) &\equiv \texttt{try}(a,\ id,\ \texttt{lift}(f)) \\ p.\texttt{then}(s, f) &\equiv \texttt{try}(a,\ \texttt{lift}(s),\ \texttt{lift}(f)) \end{aligned}$$

Any The any combinator, denoted `any`$(a_1, \ldots, a_n)$, composes n *asynchronous* arrows such that only the arrow that first emits a *progress event*, a_*, runs to completion. This combinator executes each arrow with the input of the combinator, in order, in a synchronous loop. Once arrow a_i reaches an async point, arrow a_{i+1} immediately begins execution. Because the loop running each arrow is synchronous, the event which resumes the execution of any arrow a_i will not be observed until after a_n begins listening for an event. Once some arrow a_* emits a progress event, the remaining arrows $\{a_1, \ldots, a_n\} \setminus \{a_*\}$ are canceled and the execution of a_* continues. The result of the combinator is the result of a_*.

Similar to the behavior of the `split` constructor, this arrow attempts to preclude aliasing by creating n clones of the value v.

The purpose of this combinator is to multiplex many possible external events. Synchronous arrows cannot make progress as their execution does not contain an async point. Therefore, synchronous arrows make little sense in this context and are disallowed.

This combinator is necessarily asynchronous. The first async point of the combinator is the set of first async points of each arrow a_i, which occur immediately after arrow a_n begins to yield. Once

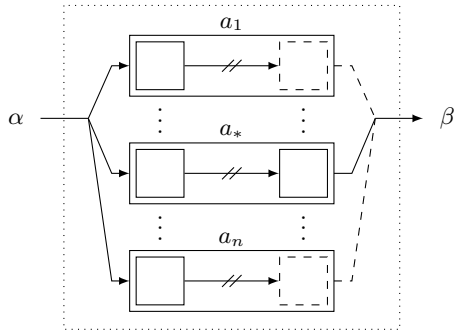

Figure 5. Dataflow diagram for the `any` combinator.

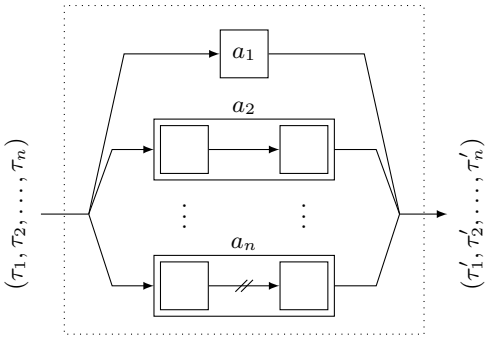

Figure 6. Dataflow diagram for the `all` combinator.

arrow a_* resumes execution, each async point of a_* is also an async point of the combinator.

The result of this combinator differs from the result of the `Promise.race` method. The former uses the value of the arrow that makes first progress where the later uses the value of the promise which rejects or resolves first. This behavior of the `any` combinator is more useful when each arrow contains multiple async points, and the progress of any of them is enough to choose a branch of execution. Then, the other arrows may be canceled to improve performance and minimize asynchronous interference.

NoEmit The no-emit combinator, denoted $\texttt{noemit}(a)$, suppresses the emission of progress events from a. This combinator creates an additional async point (and emits a progress event) after a finishes execution. Although a emits no events, it can still be preempted or canceled at its suppressed async points.

We can simulate the semantics of the `Promise.race` method (with added cancellation of *slow* arrows) by applying the `noemit` combinator to the arguments of the `any` combinator, where the arrow a_i is functionally identical to the promise p_i.

$$\texttt{race}(p_1, \ldots, p_n) \equiv \texttt{any}(\texttt{noemit}(p_1), \ldots, \texttt{noemit}(p_n))$$

The pairing of these combinators appear much more expressive than either the `any` combinator or the `Promise.race` method alone. As an example, consider two arrows representing the halves of a game, $game_1$ and $game_2$, where each arrow is composed of a non-trivial sequence of user interactions. A time-limit to the *first* portion of the game can be encoded the following.

$$\texttt{any}(\texttt{delay}(limit),\ \texttt{seq}(\texttt{noemit}(game_1),\ game_2))$$

Here, the `delay` arrow will register a listener for a timer event and immediately yield, where $game_1$ begins to execute. If the timer runs out before $game_1$ finishes, then $game_1$ is canceled at its next async point. If $game_1$ finishes before the timer runs out, then the timer is canceled and the execution path of `any` continues unobstructed towards $game_2$.

All The all combinator, denoted $\texttt{all}(a_1, \ldots, a_n)$, composes n arrows that execute concurrently. This combinator begins executing each arrow, in order, in a synchronous loop. Once arrow a_i completes or reaches an async point, arrow a_{i+1} immediately begins execution. Once all arrows have been started, they may progress through their execution in any order until they all complete, at which point the combinator completes.

The input to the combinator is an n-element tuple, where the input of each arrow a_i is the ith element of the tuple. The result of the combinator is also an n-element tuple, where the ith element of the tuple is the result of arrow a_i.

This combinator is asynchronous if *any* arrow a_i is asynchronous. The set of async points of the combinator is the union of the async points of each arrow a_i.

We can construct a combinator equivalent to the unary combinator

$$\mathit{first} : (A \rightsquigarrow B) \to (A \times C \rightsquigarrow B \times C)$$

in the arrow calculus [5] using this combinator and an *identity* arrow:

$$\mathit{first}\ a \equiv \texttt{all}(a,\ id)$$

Repeat The repeat combinator, denoted $\texttt{repeat}(a)$, executes the arrow a *at least* once. The input of the combinator is fed into a. The result of a must be a tagged union of the form

$$\{\ \texttt{"repeat"}\texttt{:}\ rep\texttt{,}\ \texttt{"value"}\texttt{:}\ val\ \}$$

where rep is either `true` or `false`. When $rep = \texttt{true}$, the combinator reinvokes itself with the value val as input. Otherwise, the combinator halts, resulting in the value val.

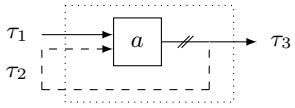

Figure 7. Dataflow diagram for the `repeat` combinator.

This combinator creates an async point following each invocation of the arrow a. This async point may progress immediately. This async point enables preemption and cancellation between iterations, and prevents synchronous arrows from looping indefinitely.

3.3 CPS Encoding

Arrows are implemented in continuation-passing style (CPS). Each arrow has an associated `call` function accepting a value argument x, a progress object p, a continuation function k, and an error handling function h. Instead of returning a value produced by the arrow, it is simply passed to k (on success) or h (on error). The progress object p is used to track async points for cancellation and emits progress events (unless suppressed) which are observed by the `any` combinator.

To demonstrate the use of the progress object p, we give the CPS encoding for the `delay` constructor in Figure 8. The `any` combinator creates a fresh progress object for each of its children. When one progress object emits a progress event, its sibling arrows are canceled. The `noemit` combinator creates a fresh progress object which does not emit events.

To demonstrate the use of the error callback h, we give the CPS encodings for the `lift` constructor and the `try` combinator in Figure 9 and Figure 10, respectively.

```
call(x, p, k, h) {
  const cancel = () => clearTimeout(timer);
  const runner = () => {
    // Emit progress event and remove canceler
    p.advance(cancel);
    k(x);
  };

  // Kick off event
  var timer = setTimeout(runner, duration);
  p.addCanceler(cancel);
}
```

Figure 8. Encoding for `delay`($duration$).

```
call(x, p, k, h) {
  try {
    // Runtime type checks and parameter "spreading"
    // sugar at this point, but omitted for brevity.
    var y = f(x);
  } catch (e) {
    return h(e); // Error continuation
  }

  k(y); // Success continuation
}
```

Figure 9. Encoding for `lift`(f) - dynamic type-checks omitted.

```
call(x, p, k, h) {
  // Invoke original error callback "h" if either
  // callback "as" or "af" creates an error value.
  // This allows nesting of error callbacks.
  a.call(x, p,
    y => as.call(y, p, k, h),
    z => af.call(z, p, k, h)
  );
}
```

Figure 10. Encoding for `try`(a, a_s, a_f).

4. Type Inference

In this section, we introduce the type system of our arrows library. We define the types of values which can be consumed or produced by arrows in Section 4.1. We define the types of arrows and give the typing rules for arrow constructors and arrow combinators in Section 4.2.

4.1 Value Types

Given a set of named types B which includes both JavaScript primitives (e.g. *Number*, *Bool*, *String*) as well as event primitives (e.g. *Elem*, *Event*), we define the type of *primitive* values, denoted b, as follows.

$$b ::= \iota \in B \mid \iota_1 + \cdots + \iota_n$$

A sum type consisting solely of named types is represented by $\iota_1 + \cdots + \iota_n$, where each ι_i is unique. The order of the types in a sum type is insignificant, and any permutation represents an equivalent type. A sum type of $n = 1$ elements is equivalent to its unique type.

Given an infinite set of type variables A, we define the types of values consumed or produced by arrows, denoted τ, as follows.

$$\begin{aligned}\tau ::= {} & b \mid \alpha, \beta \in A \mid \top \mid () \mid \langle \mathit{loop} : \tau_1,\ \mathit{halt} : \tau_2 \rangle \\ & \mid [\tau] \mid (\tau_1, \ldots, \tau_n) \mid \{\ell_1 : \tau, \ldots, \ell_n : \tau_n\}\end{aligned}$$

The *top* (*any possible*) type is represented by $\top$. The *unit* type, `()`, is fulfilled by by the Javascript value `undefined`. The *loop* type is a tagged union represented by $\langle \mathit{loop} : \tau_1,\ \mathit{halt} : \tau_2 \rangle$ used primarily by the `repeat` combinator. An arrow a produces a value v_1 of type τ_1 when it expects to be called again with v_1 as an argument. Otherwise, a produces a value v_2 of type τ_2, which is the final result of the arrow. We represent this tagged union in JavaScript as a simple object with a tag and a value field, as noted in Section 3.2.

An array type with homogeneous elements is represented by $[\tau]$, a tuple type is represented by $(t_1, \ldots, t_n)$, and a record type is represented by $\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$. The order of the labels in a record is insignificant, and any permutation of the labels represents an equivalent type.

4.2 Arrow Types

We define the types of arrows, denoted by $\widetilde{\tau}$, as follows, where C is a set of constraints of the form $\tau \leq \tau'$ and E is the set of types which may be produced in exceptional cases.

$$\widetilde{\tau} ::= \tau_{in} \rightsquigarrow \tau_{out} \setminus (C,\ E)$$

If C and E are both empty, $\tau_{in} \rightsquigarrow \tau_{out}$ may be written for short. If the constraint set C is not *consistent*, then the type is considered malformed and the associated composition is rejected during type-checking. The accompanying technical report [2] outlines an algorithm for determining whether a constraint set is consistent. In brief, the algorithm rejects constraint sets whose transitive closure contains obvious subtyping violations.

The constrained arrow type is similar to the constrained type $\tau \setminus C$ introduced by Eifrig et al. [1], where the set C contains subtyping constraints on the type variables occurring in τ. A constrained type inference system generalizes unification-based inference to languages with subtyping - a feature we found is necessary for arrow type inference.

We assume that if a constrained arrow type contains a type variable α in τ_{in}, τ_{out}, C, or E, that the type variable is understood to be universally quantified with respect to the arrow type, i.e.

$$\forall \alpha . \tau_{in} \rightsquigarrow \tau_{out} \setminus (C,\ E)$$

Typing rules for arrow constructors and combinators appear in Figure 11 and Figure 12, respectively. For brevity, the typing rules have the implicit assumption that if $a : \tau \rightsquigarrow \tau' \setminus (C,\ E)$, then C is consistent.

When an arrow type is used as the input of a combinator, a unique instantiation of that type is created in order to prevent unintended clashing of type variables. A unique instantiation of a constrained arrow type is created by substituting the set of type variables occurring in the type as well as the constraint set and set of error types with a set of fresh type variables.

Rule (T-LIFT) assumes that each lifted function f is annotated with a constrained arrow type describing the input and output types of f, and Rule (T-AJAX) assumes that each Ajax configuration function c is annotated with two constrained types: one describing the input to c, and one describing the response from the remote server. We assume the existence of an implicit function `Annot`(t, f) which reads the annotation named t of the function f and produces a unique instantiation of the type it describes.

Rule (T-NTH) shows how the `nth`(n) combinator selects the nth element from a tuple with $m \geq n$ elements. The argument to this combinator may be a *wider* tuple, as $(\tau_1, \ldots, \tau_m) \leq (\tau'_1, \ldots, \tau'_n)$ is a consistent constraint. Note that the application of this rule happens at arrow composition time when n is known.

5. Discussion

We have implemented several small but non-trivial programs using the abstractions provided by our library with type-checking enabled during development. Among these were an implementation for the game *Memory*, which requires the user to select two cards from a grid with the same face value until all pairs of cards are selected, and an application which demonstrates *Fischer-Yates Shuffle* and *Bubble Sort* algorithms through timed animations.

$$\text{T-LIFT}\quad \frac{\texttt{Annot}(arrow,\ f) = \tau_1 \rightsquigarrow \tau_2 \setminus (C,\ E)}{\texttt{lift}(f) : \tau_1 \rightsquigarrow \tau_2 \setminus (C,\ E)}$$

$$\text{T-AJAX}\quad \frac{\texttt{Annot}(conf,\ c) = \tau_1 \setminus (C_1, E) \qquad \texttt{Annot}(resp,\ c) = \tau_2 \setminus C_2}{\texttt{ajax}(c) : \tau_1 \rightsquigarrow \tau_2 \setminus (C_1 \cup C_2,\ E \cup \{AjaxError\})}$$

$$\text{T-ELEM}\quad \texttt{elem}(selector) : \top \rightsquigarrow Elem$$

$$\text{T-EVENT}\quad \texttt{event}(name) : Elem \rightsquigarrow Event$$

$$\text{T-DELAY}\quad \texttt{delay}(duration) : \alpha \rightsquigarrow \alpha$$

$$\text{T-SPLIT}\quad \texttt{split}(n) : \alpha \rightsquigarrow \underbrace{(\alpha, \ldots, \alpha)}_{n \text{ elements}}$$

$$\text{T-NTH}\quad \texttt{nth}(n) : \underbrace{(\alpha, \beta, \ldots, \gamma)}_{n \text{ elements}} \rightsquigarrow \gamma$$

Figure 11. Typing rules for arrow constructors.

$$\text{T-REPEAT}\quad \frac{a : \tau_1 \rightsquigarrow \langle loop : \tau_2,\ halt : \tau_3 \rangle \setminus (C,\ E) \qquad C' = \{\tau_2 \leq \tau_1\}}{\texttt{repeat}(a) : \tau_1 \rightsquigarrow \tau_3 \setminus (C \cup C',\ E)}$$

$$\text{T-SEQ}\quad \frac{\forall i \in 1..n.\ a_i : \tau_i \rightsquigarrow \tau_i' \setminus (C_i,\ E_i) \qquad C' = \cup_{i=2}^{n} \{\tau_{i-1}' \leq \tau_i\}}{\texttt{seq}(a_1, \ldots, a_n) : \tau_1 \rightsquigarrow \tau_n' \setminus (C' \cup \bigcup C_i,\ \bigcup E_i)}$$

$$\text{T-ALL}\quad \frac{\forall i \in 1..n.\ a_i : \tau_i \rightsquigarrow \tau_i' \setminus (C_i,\ E_i)}{\texttt{all}(a_1, \ldots, a_n) : (\tau_1, \ldots, \tau_n) \rightsquigarrow (\tau_1', \ldots, \tau_n') \setminus (\bigcup C_i,\ \bigcup E_i)}$$

$$\text{T-ANY}\quad \frac{\forall i \in 1..n.\ a_i : \tau_i \rightsquigarrow \tau_i' \setminus (C_i,\ E_i),\ C_i' = \{\alpha \leq \tau_i,\ \tau_i' \leq \beta\}}{\texttt{any}(a_1, \ldots, a_n) : \alpha \rightsquigarrow \beta \setminus (\bigcup C_i' \cup \bigcup C_i,\ \bigcup E_i)}$$

$$\text{T-NOEMIT}\quad \frac{a : \widetilde{\tau}}{\texttt{noemit}(a) : \widetilde{\tau}}$$

$$\text{T-TRY}\quad \frac{\forall i \in 1..3.\ a_i : \tau_i \rightsquigarrow \tau_i' \setminus (C_i,\ E_i) \qquad C' = \{\tau_1' \leq \tau_2,\ \tau_2' \leq \beta,\ \tau_3' \leq \beta\} \cup \{\tau \leq \tau_3 \mid \tau \in E_1\}}{\texttt{try}(a_1, a_2, a_3) : \tau_1 \rightsquigarrow \beta \setminus (C' \cup \bigcup C_i,\ E_2 \cup E_3)}$$

Figure 12. Typing rules for arrow combinators.

During this time, we observed a large number of instances where the type system forbid us from composing arrows illegally. In many cases the composition was illegal in a way that was trivial to fix yet non-obvious to discover. For example, our game *Memory* used an arrow with the type $selectOne :: Elem \rightsquigarrow ()$, which was meant to be executed twice in a row with the same input. The intuition when composing such arrows is to simply `seq` them together, but this unfortunately causes a type clash between the first and second invocations. The correct solution is to *remember* the input to the first invocation, and use it as the input of the second invocation. This became a common idiom and was encoded as a derived combinator in our API.

$$\texttt{remember}(a) \equiv \texttt{seq}(\texttt{split}(2),\ \texttt{all}(a,\ id),\ \texttt{nth}(2))$$

Annotation Burden The annotation burden required by developers seems to be minimal. Our implementation of *Memory* (151 lines of ES6) required only eight annotations in total, but our type-checker inferred the type of 126 arrows at startup. This number is not surprising if we consider that many combinators (such as the `remember` combinator defined above) are built from the *foundational* combinators discussed in Section 3. Similarly, our implementation for the sorting and shuffling animation (126 lines of ES6) required only four annotations, but types for 124 arrows were inferred.

Inference Overhead We measured the runtime overhead of arrow type inference. We used the Ajax example from Section 2 as well as the programs described in this section. Measurements are averaged over 1000 runs in Chrome (V8), with warmup runs discarded.

Application	# Arrows	Disabled (ms)	Enabled (ms)
Ajax	13	0.125	0.905
Shuffle	62	0.837	3.957
Shuffle & Sort	124	1.638	8.118
Memory Game	126	1.566	10.989

We also constructed a benchmark application which allows us to arbitrarily adjust the size of arrow types. We used an arrow with a type of the form

$$\{\overline{f_i : \alpha_i}\} \rightsquigarrow \{\overline{f_i : \alpha_i'}\} \setminus (\{\overline{\alpha_i \leq \alpha_i'}\})$$

and composed it with itself it 1000 times. This requires inferring a large number of intermediate arrow types, each with a constraint set size linear to the size of the its input.

We measured the runtime overhead of arrow type inference with a variable number of fields in the arrow's type. Based on these results, it appears that arrow type inference is linear with the number of arrows and subquadratic with the size of the arrow type. We expect arrow type sizes to remain small as the user annotates only the fields of the object which are used by the arrow, and arrows types are aggressively simplified during inference.

# Fields	1	10	20	30	40	50
Time (ms)	1.10	3.39	6.88	11.82	17.32	25.48

We measured the overhead of the runtime type-checks at the border of lifted functions. Without runtime type-checks, the arrow executes in an invariant $1.762ms$. There are 1024 dynamic type checks (the number of lifted arrows) performed in a single run. Based on these results, it appears that runtime type-checks have an overhead which grows linearly with both the number of dynamic type checks as well as the size of the *type* being traversed.

# Fields	1	10	20	30	40	50
Overhead (ms)	0.25	1.24	2.11	3.07	4.06	4.81

It is important to keep in mind that application using these abstractions are asynchronous and often blocked waiting for user or remote server responses, which vastly dominate the runtime of an application. We find this performance overhead during development to be negligible.

6. Related Work

Arrows Arrows [5, 7, 8] were first formalized as a generalization of monads [11]. An arrow of type $(a\ b\ c)$ represents a computation with input of type b delivering a value of type c. Our `lift` constructor and the combinators `seq` and `all` encompass the three operations which define arrows.

Arrowlets Arrowlets is a JavaScript library for using arrows [6], providing programs the means to elegantly structure event-driven web components that are easy to understand, modify, and reuse. The implementation of our arrows library was heavily inspired by the continuation-passing style used by Arrowlets, as well as the asynchronous semantics of the combinators it provides.

Regarding execution semantics only, there are two major differences between our arrows library and Arrowlets. First, we have generalized *binary* combinators to support n arrows, leading to code which favors *generalized n-tuples* over simple pairs. Second, we have altered the encoding of arrows to carry along an error continuation in addition to the normal-path continuation. This allowed us to add the `try` combinator, which subsumes the semantics of ES6 Promises.

ES6 Promises Promises allow a sequence of callbacks to be chained together, flattening the dreaded 'pyramid of doom' into a sequence of promise `then` calls. Promises also provide a means of error handling, where the `then` method accepts an optional error callback.

Our arrows library also encode the core mechanism of promises, but there are some obvious differences in execution semantics. For one, when a promise object is created it attempts to resolve immediately. If a promise object is composed with a callback after its resolution, it simply forwards the memoized result. Arrows separate composition and execution behind an explicit `run` method. This allows an arrow to be called multiple times, like a regular function, and enables features such as the `repeat` combinator. Promises place emphasis on the values which they *proxy*, where arrows place emphasis on the *computation*. It would be trivial to adapt our arrows library to support the *lazy* nature of Promises with the addition of a memoizing combinator.

Promises also implement two methods which are strongly related to the arrow combinators presented here. The method `Promise.all`(ps), similar to the `all` combinator, takes an iterable of promises, ps, and resolves once each promise resolves or rejects if any promise rejects. Its resolved value is an array of the resolved values of each promise. The method `Promise.race`(ps), similar to the `any` combinator when the arrow inputs are wrapped in `noemit`, takes an iterable of promises, ps, and resolves once *any* promise p resolves or rejects once *any* promise p rejects. The value of the promise is the value of the first resolved arrow. Unlike the `any` combinator, `Promise.race` does not abort the execution of the remaining arrows. We believe the semantics of the `any` combinator to be more useful in practice.

Coincidently, because we can simulate promise semantics so closely with arrows, our typing judgments can also apply almost directly to a promise library. However, type-checking with Arrows is much more elegant than with Promises because the composition time and execution time of an arrow has a clear delineation, where a promise may begin immediately following its creation.

Promises and Arrowlets attack the problem of callback composition in similar ways, but provide a disjoint set of orthogonal features. Arrowlets provide a means to abort an asynchronous operation, where Promises follow a fire-and-forget convention. Promises provide a means of catching an error, where Arrowlets focus only on happy-path composition. Our implementation of arrows chooses to support both sets of features.

Factors Factors [10] are another interactivity abstraction. A factor represents a state of a program which can be *queried* either synchronously or asynchronously. A synchronous query takes a *prompt* value and blocks until a *response* value is produced. An asynchronous query takes a *prompt* value and returns immediately, but produces a *future factor* which serves as a handle of the computation. Because queries return a *continuation* factor, state is explicitly tracked. Factors require an affine type system to ensure that future factors are not used more than once.

7. Conclusion

We have presented an arrows library which encodes sematics similar to ES6 Promises and a composition-time type-checker which enables type-directed development. We believe this tool greatly reduces the friction of development using a functional style in a language with no compile-time checks.

Future Work We intend to explore additional methods of 'static' (composition-time) analysis to provide greater confidence in correct arrow compositions. We are currently exploring state analysis with interesting results.

Acknowledgments

We thank John Boyland for his comments on the draft.

References

[1] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to oop. *Electronic Notes in Theoretical Computer Science*, 1:132–153, 1995.

[2] E. Fritz and T. Zhao. Type inference of asynchronous arrows in JavaScript. Technical report, University of Wisconsin - Milwaukee, 2015.

[3] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP 2010–Object-Oriented Programming*, pages 126–150. Springer, 2010.

[4] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

[5] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

[6] Y. P. Khoo, M. Hicks, J. S. Foster, and V. Sazawal. Directing JavaScript with arrows. In *Proceedings of the 5th Symposium on Dynamic Languages*, DLS '09, pages 49–58, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-769-1.

[7] S. Lindley, P. Wadler, and J. Yallop. The arrow calculus. *Journal of Functional Programming*, 20(01):51–69, 2010.

[8] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 229(5):97–117, 2011.

[9] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Programming languages and systems*, pages 307–325. Springer, 2008.

[10] S. K. Muller, W. A. Duff, and U. A. Acar. Practical abstractions for concurrent interactive programming. Technical report, Carnegie Mellon University, 2015.

[11] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.

[12] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Notices*, volume 35, pages 242–252. ACM, 2000.