Inferring Types for 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].

[1] https://pantherfile.uwm.edu/fritz/www/arrows

```
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, generally 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 event (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 my mutable arrows.

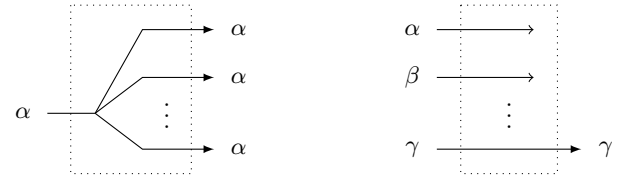

Figure 2. Dataflow diagrams for `split` and `nth` arrows.

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 can 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.

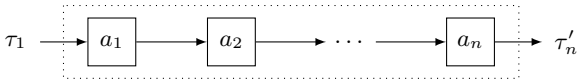

Figure 3. Dataflow diagram for the `seq` combinator.

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.

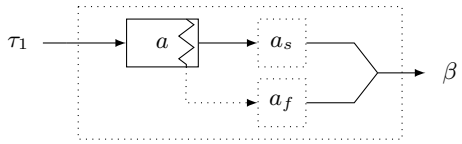

Figure 4. Dataflow diagram for the `try` combinator.

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

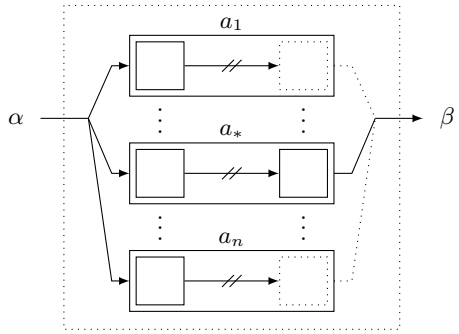

Figure 5. Dataflow diagram for the `any` combinator.

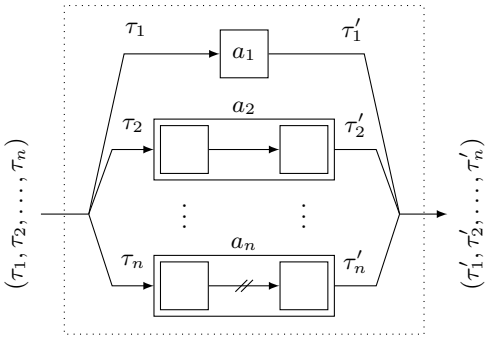

Figure 6. Dataflow diagram for the `all` combinator.

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 Promise's `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 sync point (and emits a progress event) after a finishes execution. Although a emits no events, it can still be preempted or canceled at its async points.

We can simulate the semantics of Promise's `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 Promise's `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.

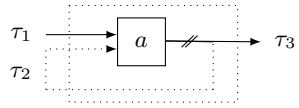

Figure 7. Dataflow diagram for the `repeat` combinator.

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.

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

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 $\langle loop : \tau_1,\ halt : \tau_2 \rangle$ 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 appendix (Sections A and B) 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 existentially 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 $selectOne :: Elem \rightsquigarrow \top$, 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 you consider 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 Promises semantics so closely with arrows, our typing judgments can also apply almost directly to a Promises library. However, type-checking with Arrows is much more elegant than with Promises because the composition time and execution time of arrows has a clear delineation, where Promises may begin immediately following their creation.

Promise 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 analysis to provide greater confidence in correct arrow compositions. We are currently exploring typestate analysis with relation to the semantics of concurrently executing arrows 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. Inferring types for 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.

$$\textsc{Cls-Trans}\ \frac{\tau_1 \le \tau_2 \in C \qquad \tau_2 \le \tau_3 \in C}{\tau_1 \le \tau_3 \in C} \qquad \textsc{Cls-Loop}\ \frac{\langle \mathit{loop} : \tau_1,\ \mathit{halt} : \tau_2\rangle \le \langle \mathit{loop} : \tau_1',\ \mathit{halt} : \tau_2'\rangle \in C}{\{\tau_1 \le \tau_1', \tau_2 \le \tau_2'\} \subseteq C} \qquad \textsc{Cls-Array}\ \frac{[\tau] \le [\tau'] \in C}{\tau \le \tau' \in C}$$

$$\textsc{Cls-Tuple}\ \frac{(\tau_1, \ldots, \tau_n) \le (\tau_1', \ldots, \tau_k') \in C}{\{\tau_i \le \tau_i' \mid i \in 1..n\} \subseteq C} \qquad \textsc{Cls-Record}\ \frac{\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \le \{\ell_1 : \tau_1', \ldots, \ell_k : \tau_k'\} \in C}{\{\tau_i \le \tau_i' \mid i \in 1..k\} \subseteq C}$$

Figure 13. Constraint set closure rules.

$$\textsc{Cns-Top}\ \frac{}{\tau \le \top} \qquad \textsc{Cns-Var}\ \frac{\tau \in A \vee \tau' \in A}{\tau \le \tau'} \qquad \textsc{Cns-Sum}\ \frac{\{\iota_i \mid i \in 1..k\} \subseteq \{\iota_i' \mid i \in 1..n\}}{\iota_1 + \cdots + \iota_n \le \iota_1' + \cdots + \iota_k'} \qquad \textsc{Cns-Array}\ \frac{}{[\tau] \le [\tau']}$$

$$\textsc{Cns-Loop}\ \frac{}{\langle \mathit{loop} : \tau_1,\ \mathit{halt} : \tau_2\rangle \le \langle \mathit{loop} : \tau_1',\ \mathit{halt} : \tau_2'\rangle} \qquad \textsc{Cns-Tuple}\ \frac{k \ge n}{(\tau_1, \ldots, \tau_k) \le (\tau_1', \ldots, \tau_n')} \qquad \textsc{Cns-Record}\ \frac{\{\ell_i' \mid i \in 1..n\} \subseteq \{\ell_i \mid i \in 1..k\}}{\{\ell_1 : \tau_1, \ldots, \ell_k : \tau_k\} \le \{\ell_1' : \tau_1, \ldots, \ell_n' : \tau_n'\}}$$

Figure 14. Constraint set consistency rules.

A. Consistency

In this section we present the definition of constraint set consistency. If an arrow has the type $\tau \rightsquigarrow \tau' \setminus (C,\ E)$ and C is *inconsistent*, then there is either some type variable α which has a set of unsatisfiable bounds, or there is a type clash between the output and input of two child arrows.

Definition A.1 (Closed)**.** A set of constraints C is *closed* if it satisfies the closure rules given in Figure 13. We refer to the closure of C as $\mathit{closure}(C)$.

Definition A.2 (Consistent)**.** A constraint set C is *consistent* if every constraint in $\mathit{closure}(C)$ is consistent. A consistent constraint must match one of the forms given in Figure 14.

Rule (CLS-TRANS) ensures that subtype constraints are transitive. For example, $C = \{\mathit{String} \le \alpha, \alpha \le \mathit{Number}\}$ contains only consistent constraints. However, due to the unsatisfiable bounds of α, $\mathit{String} \le \mathit{Number} \in \mathit{closure}(C)$ and the constraint set is considered inconsistent.

The remaining closure and consistency rules describe a simple subtyping join-semilattice. The *top* type occupies the top of the lattice, by Rule (CNS-TOP); there is no bottom type (and hence no greatest lower bound for some sets of types). This allows an arrow consuming *any* type to be composed with any arrow, which is a useful property when sequencing.

Named types are neither subtypes nor supertypes of another named type. Named types are subtypes of any sum type which contains them, and sum types are subtypes of their own supersets, by Rule (CNS-SUM). This allows an arrow producing a set of types T and an arrow consuming a set of types T′ to be composed when $\mathrm{T} \subseteq \mathrm{T}'$.

Rules (CNS-LOOP), (CNS-ARRAY), (CNS-TUPLE), and (CNS-RECORD) ensure that composite datatypes are consistent only with composite datatypes with the same outermost type constructor. Tuple and record width subtyping is enabled by Rules (CNS-TUPLE) and (CNS-RECORD). Array, tuple, and record depth subtyping is enabled by Rules (CLS-LOOP), (CLS-ARRAY), (CLS-TUPLE), and (CLS-RECORD).

Type variables are never immediately inconsistent with another type, by Rule (CNS-VAR). This makes it possible to have a set of constraints describing an impossible lower bound for some type variable α (e.g. $C = \{\alpha \le \mathit{String}, \alpha \le \mathit{Number}\}$, where no lower bound of both String and Number exists). This case is handled by type simplification, discussed in Section B.

B. Type Simplification

In this section we present a *type simplification* technique for arrow types. Our goal is to remove as many *unnecessary* constraints from the constraint set of an arrow type as possible. This keeps the size of arrow types small, decreasing memory overhead and runtime overhead. Without simplification, arrow types are noticeably larger and type inference is noticeably slower as closure calculation and consistency checks are at least linear with the size of the arrow type.

As a motivating example, consider the following composition involving an arrow a of type $\mathit{Event} \rightsquigarrow \top$.

$$\texttt{remember}(\texttt{seq}(\texttt{event}(\mathit{click}),\ a))$$

The resulting arrow takes as input an Elem value, waits until a *click* events occurs on that value, invokes the arrow a with the resulting event, and then yields the original Elem value. This is useful as it allows multiple events to be sequenced on the same event.

Without type simplification, the type of the resulting arrow is given by the following.

$$\alpha \rightsquigarrow \delta \setminus (\{\ \mathit{Event} \le \mathit{Event},\ (\alpha,\ \alpha) \le (\mathit{Elem},\ \beta),\ (\top,\ \beta) \le (\gamma,\ \delta),\ \alpha \le \mathit{Elem},\ \alpha \le \beta,\ \top \le \gamma,\ \beta \le \delta,\ \alpha \le \delta\ \}, \emptyset)$$

The first constraint, $\mathit{Event} \le \mathit{Event}$, is introduced by the inner `seq`. The next two constraints $(\alpha,\ \alpha) \le (\mathit{Elem},\ \beta)$ and $(\top,\ \beta) \le (\gamma,\ \delta)$, are introduced by the `remember` combinator, which involves a `seq` with an `split` and `nth` arrow. The remaining constraints are introduced by closure rules described in Section A.

After type simplification we are given the following, which is much smaller and more easily understandable.

$$\alpha \rightsquigarrow \delta \setminus (\{\ \alpha \le \delta,\ \alpha \le \mathit{Elem}\ \}, \emptyset)$$

The *Memory* application mentioned in Section 5 creates an arrow with 1414 distinct constraints involving 56 type variables with type simplification disabled, and **no** constraints after minimization.

ULS-TOP
$\tau \leq \top$

ULS-SELF
$\tau \leq \tau$

ULS-UPPER
$\alpha^+ \leq \tau$

ULS-LOWER
$\tau \leq \alpha^-$

ULS-NONVAR
$$\frac{\tau \notin A \wedge \tau' \notin A}{\tau \leq \tau'}$$

ULS-LOWERUNKNOWN
$$\frac{\alpha \notin \tau_{in} \wedge \alpha \notin \tau_{out}}{\alpha \leq \beta}$$

ULS-UPPERUNKNOWN
$$\frac{\alpha \notin \tau_{in} \wedge \alpha \notin \tau_{out}}{\beta \leq \alpha}$$

Figure 15. Useless constraint elimination rules.

Definition B.1 (Simplified)**.** An arrow type is *simplified* if it is *bound-minimal* (Definition B.2), *variable-minimal* (Definition B.4), and *pruned* (Definition B.5).

Definition B.2 (Bound Minimal)**.** An arrow type $\tau_{in} \rightsquigarrow \tau_{out} \setminus (C)$ is *bound-minimal* if every type variable α in the arrow type has at most one *concrete upper bound* and at most one *concrete lower bound*. A type is *concrete* if it contains no type variables. We can *bound-minimize* an arrow type by *collapsing* the concrete bounds of a type variable.

We can collect the concrete lower and upper bounds of a type variable α, denoted $b_\downarrow(\alpha)$ and $b_\uparrow(\alpha)$, respectively.

$$b_\downarrow(\alpha) = \{\tau_i \mid \tau_i \leq \alpha \in C \text{ and } \tau_i \text{ contains no type variables}\}$$
$$b_\uparrow(\alpha) = \{\tau_i \mid \alpha \leq \tau_i \in C \text{ and } \tau_i \text{ contains no type variables}\}$$

We can then *bound-minimize* a constraint set C by collapsing the lower and upper bounds for each type variable α. We can collapse the lower bounds of a type variable α by applying the following transformation to C.

$$(C \setminus \{\tau_i \leq \alpha \mid \tau_i \in b_\downarrow(\alpha)\}) \cup \{(\vee\, b_\downarrow(\alpha)) \leq \alpha\}$$

Similarly, we can collapse the upper bounds of a type variable α by applying the following transformation to C.

$$(C \setminus \{\alpha \leq \tau_i \mid \tau_i \in b_\uparrow(\alpha)\}) \cup \{\alpha \leq (\wedge\, b_\uparrow(\alpha))\}$$

$(\vee\ \mathrm{T})$ and $(\wedge\ \mathrm{T})$ denote the least upper bound and greatest lower bound of the set of types T, respectively. An upper bound necessarily exists between any two concrete types due to the presence $\top$, but a lower bound may not exist due to the absence of a bottom type.

If a non-existent lower bound is needed to simplify an arrow type, then there is a type variable α for which no concrete type satisfying the set of constraints exists. We consider such arrow types *malformed*. For example, an arrow of the following type accepts an (impossible) value whose type must be *simultaneously* a lower bound of *Int* and a lower bound of *String*.

$$\alpha \rightsquigarrow \mathit{Number} \setminus (\{\alpha \leq \mathit{Int}, \alpha \leq \mathit{String}\},\ \emptyset)$$

Such an arrow type, while consistent, results in a composition error as it cannot be supplied any reasonable value at runtime.

Definition B.3 (Type Variable Position)**.** A type variable α may occur in *negative position*, denoted α^-, in *positive position*, denoted α^+, in both positions simultaneously, denoted $\alpha^\pm$, or in neither position, denoted α, relative to an arrow type $\tau_{in} \rightsquigarrow \tau_{out} \setminus (C,\ E)$.

Given a constraint $\tau \leq \tau'$ and a type variable α occurring in τ and a type variable β occurring in τ', we say that α *lower-bounds* β and β *upper-bounds* α.

A type variable α occurs in *negative position* if either α occurs in τ_{in} or if α upper-bounds some type variable β^- or $\beta^\pm$. Symmetrically, a type variable α occurs in *positive position* if α occurs in τ_{out} or if α lower-bounds some type variable β^+ or $\beta^\pm$.

Definition B.4 (Variable-Minimal)**.** We can *variable-minimize* an arrow type $\tau_{in} \rightsquigarrow \tau_{out} \setminus (C,\ E)$ by constructing a substitution $\sigma = [\tau_i/\alpha_i]$ by the rules below and replacing all occurrences of the tyep variable α_i by the type τ_i in τ_{in}, τ_{out}, C, and E. An arrow type is *variable-minimal* if no such substitution can be created.

We add the mapping $[\tau/\alpha]$ to the substitution σ if one of the following rules hold.

– $\{\tau \leq \alpha,\ \alpha \leq \tau\} \subseteq C$
– $\alpha^- \leq \tau \in C$ and $\forall(\tau' \neq \tau),\ \alpha \leq \tau' \notin C$
– $\tau \leq \alpha^+ \in C$ and $\forall(\tau' \neq \tau),\ \tau' \leq \alpha \notin C$

If $[\beta/\alpha]$ is being added to a substitution σ which already contains the mapping $[\tau/\alpha]$, then we instead add the mapping $[\tau/\beta]$ to avoid re-introducing a type variable being substituted.

We substitue a type variable α with the type τ if τ is both an upper and lower bound of α, as $\alpha = \tau$ is a necessary condition for a solution to the constraint set.

We substitute type variable in negative position with their sole upper bound, and type variables in positive position with their sole lower bound. Negative position variables represent a constraint on the *input* of an arrow, as positive position variables represent a constraint on the *output* of an arrow. Therefore, negative position variables are concerned only with an upper bound, and positive position variables are concerned only with a lower bound.

Applying a substitution may alter the closure or consistency properties of a set of constraints and may require the closure set to be recalculated and the consistency rechecked.

Definition B.5 (Pruned)**.** An arrow type $\tau_{in} \rightsquigarrow \tau_{out} \setminus (C,\ E)$ is *pruned* if every constraint $c \in C$ is not immediately *useless*. Constraints matching a form in Figure 15 are considered useless. A constraint set can be *pruned* by repeatedly removing all useless constraints.

C. Semantics

In this section, we explain the semantics of arrow constructors and combinators. Section C.1 and Section C.2 defines the syntax of abstract arrows and a translation from concrete arrows to the abstract arrows. Section C.3 defines an operational semantics for abstract arrows. Section C.4 sketches a proof that the arrow type system presented in Section 4 is sound.

C.1 Syntax

Figure 16 defines the complete abstract syntax. We translate concrete arrows, denoted a, to abstract arrows, denoted e_a, of the form

$$\lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ e$$

where x is replaced by the input value of the arrow, p is replaced by a list of progress objects (or progress list), denoted e_p, k is replaced by a continuation function, and h is replaced by an exception handler function.

Progress objects, denoted P_i, always come in pairs, denoted P_i^1 and P_i^2, which are used to tag the branches of the `any` combinator. A progress list, denoted e_p, represents the path to a node in a binary tree, where e_p is the parent of $P_i^k :: e_p$. Each arrow carries a progress list throughout its execution. The expression `advance` e_p cancels the execution of the arrows carrying progress lists that are

$$
\begin{array}{rcll}
e & ::= & x & \\
& | & f & \text{function value} \\
& | & e_a \mid e_e \mid e_k \mid e_p \mid e_l \mid e_h & \\
& | & e_1\ e_2 & \text{function call} \\
& | & e_1;\ e_2 & \text{sequence} \\
& | & (e, e') \mid e[n] & \text{tuple \& tuple projection} \\
& | & \textbf{case}\ e_l\ \textbf{of}\ \texttt{loop}(x_1) \Rightarrow e_1 & \\
& & \quad \mid \texttt{halt}(x_2) \Rightarrow e_2 & \\
& | & \texttt{fix}(\lambda x.a) & \\
& | & e_1 \bullet (e_2,\ e_p,\ e_k,\ e_h) & \text{arrow application} \\
& | & \texttt{advance}\ e_p & \\
& | & \texttt{async}\ e_e\ e_p\ (\lambda y.\ e)\ (\lambda z.\ e') &
\end{array}
$$

$$
\begin{array}{rcll}
v, n & & & \text{value \& constant} \\
v_e & ::= & \texttt{timeEvent}(n) \mid \texttt{ajaxEvent}(c) & \text{event value} \\
v_p & ::= & \epsilon \mid P_i^j :: v_p & \text{progress value} \\
e_a & ::= & \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ e & \text{abstract arrow} \\
e_e & ::= & \texttt{timeEvent}(e) \mid \texttt{ajaxEvent}(e) & \text{event expression} \\
e_k & ::= & k \mid \lambda y.\ e & \text{continuation} \\
e_h & ::= & h \mid \lambda y.\ e & \text{exception handler} \\
e_p & ::= & p \mid P_i^j :: p \mid v_p & \text{progress expression} \\
e_l & ::= & x \mid \texttt{loop}(e) \mid \texttt{halt}(e) & \text{loop expression} \\
\Delta & ::= & \epsilon & \text{event context} \\
& | & \Delta, v_e \mapsto (v_p,\ \lambda y.\ e,\ \lambda z.\ e') &
\end{array}
$$

Figure 16. Abstract syntax.

$$
\begin{array}{rcl}
[\![\texttt{lift}(f)]\!] & \equiv & \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ k\ (f\ x) \\
[\![\texttt{ajax}(c)]\!] & \equiv & \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ \texttt{async}\ \texttt{ajaxEvent}(c\ x)\ p\ (\lambda y.\ \texttt{advance}\ p;\ (k\ y))\ (\lambda z.\ h(z)) \\
[\![\texttt{delay}(duration)]\!] & \equiv & \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ \texttt{async}\ \texttt{timeEvent}(duration)\ p\ (\lambda\_.\ \texttt{advance}\ p;\ (k\ x))\ (\lambda\_.\ ()) \\
[\![\texttt{repeat}(a)]\!] & \equiv & \texttt{fix}(\lambda r. \\
 & & \quad \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ [\![a]\!] \bullet (x,\ p,\ \lambda y. \\
 & & \quad\quad \texttt{advance}\ p; \\
 & & \quad\quad \textbf{case}\ y\ \textbf{of}\ \texttt{loop}(z) \Rightarrow \texttt{async}\ \texttt{timeEvent}(0)\ p\ (\lambda\_.\ r \bullet (z,\ p,\ k,\ h))\ h) \\
 & & \quad\quad\quad \mid \texttt{halt}(z) \Rightarrow (k\ z),\ h) \\
[\![\texttt{seq}(a_1, a_2)]\!] & \equiv & \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ [\![a_1]\!] \bullet (x,\ p,\ \lambda y.\ [\![a_2]\!] \bullet (y,\ p,\ k,\ h),\ h) \\
[\![\texttt{all}(a_1, a_2)]\!] & \equiv & \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ [\![a_1]\!] \bullet (x[0],\ p,\ \lambda y.\ [\![a_2]\!] \bullet (x[1],\ p,\ \lambda z.\ (k\ (y,\ z)),\ h),\ h) \\
[\![\texttt{any}(a_1, a_2)]\!] & \equiv & \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ ([\![a_1]\!] \bullet (x,\ P_i^1 :: p,\ k,\ h));\ ([\![a_2]\!] \bullet (x,\ P_i^2 :: p,\ k,\ h)) \qquad P_i^1, P_i^2 \text{ are fresh} \\
[\![\texttt{try}(a, a_s, a_f)]\!] & \equiv & \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ [\![a]\!] \bullet (x,\ p,\ \lambda y.\ [\![a_s]\!] \bullet (y,\ p,\ k,\ h),\ \lambda z.\ [\![a_f]\!] \bullet (z,\ p,\ k,\ h)) \\
[\![a.\texttt{run}()]\!] & \equiv & [\![a]\!] \bullet ((),\ \epsilon,\ \lambda\_.\ (),\ \lambda\_.\ ())
\end{array}
$$

Figure 17. Arrow translation rules.

E-ASYNC
$$\Delta, \texttt{async}\ v_e\ v_p\ (\lambda y.\ e)\ (\lambda z.\ e') \to \Delta \cup \{v_e \mapsto (v_p,\ \lambda y.\ e,\ \lambda z.\ e')\}, ()$$

E-ADVANCE-EMPTY
$$\Delta, \texttt{advance}\ \epsilon \to \Delta, ()$$

E-EVENT-SUCC
$$\frac{\begin{array}{c} v_e \mapsto (v_p,\ \lambda y.\ e,\ \_) \in \Delta \\ Resp(v_e) = Succ(v) \\ \emptyset \vdash v : \tau_1 \qquad \emptyset \vdash v_e : \langle succ : \tau_1, fail : \_ \rangle \end{array}}{\Delta, () \to \Delta \setminus \{v_e \mapsto (v_p,\ \lambda y.\ e,\ \_)\}, [v/y]e}$$

E-EVENT-FAIL
$$\frac{\begin{array}{c} v_e \mapsto (v_p,\ \_,\ \lambda z.\ e') \in \Delta \\ Resp(v_e) = Fail(v) \\ \emptyset \vdash v : \tau_2 \qquad \emptyset \vdash v_e : \langle succ : \_, fail : \tau_2 \rangle \end{array}}{\Delta, () \to \Delta \setminus \{v_e \mapsto (v_p,\ \_,\ \lambda z.\ e')\}, [v/z]e'}$$

E-ADVANCE
$$\frac{\Delta' = \{v_e \mapsto (v'_p,\ \_,\ \_) \in \Delta \mid P_i^k \notin v'_p,\ k \neq j\}}{\Delta, \texttt{advance}\ (P_i^j :: v_p) \to \Delta', \texttt{advance}\ v_p}$$

E-HOST-APP
$$\frac{\begin{array}{c} f\ v \downarrow v' \qquad \texttt{annot}_{arrow}(f) = \tau_1 \to \tau_2 \setminus C \\ \emptyset \vdash v : \tau_1 \setminus C_1 \qquad \emptyset \vdash v' : \tau_2 \setminus C_2 \qquad \texttt{Cl}(C_2) \subseteq \texttt{Cl}(C \cup C_1) \end{array}}{\Delta, f\ v \to \Delta, v'}$$

E-ARROW-APP
$$\frac{e_a = \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ e_0}{\Delta, e_a \bullet (v,\ v_p,\ \lambda y.e,\ \lambda z.e') \to \Delta, [v/x, v_p/p, \lambda y.e/k, \lambda z.e'/h]e_0}$$

Figure 18. Operational semantics.

T-TIME
$$\frac{\Gamma \vdash e : Number}{\Gamma \vdash \texttt{timeEvent}(e) : \langle succ : (), fail : () \rangle}$$

T-AJAX
$$\frac{\Gamma \vdash e : AjaxConf}{\Gamma \vdash \texttt{ajaxEvent}(e) : \langle succ : \texttt{annot}_{resp}(e), fail : AjaxErr \rangle}$$

T-SUB
$$\frac{\Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau \setminus \{\tau' \leq \tau\}}$$

T-LOOP
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{loop}(e) : \langle loop : \tau,\ halt : \tau' \rangle}$$

T-HALT
$$\frac{\Gamma \vdash e : \tau'}{\Gamma \vdash \texttt{halt}(e) : \langle loop : \tau,\ halt : \tau' \rangle}$$

T-ADVANCE
$$\frac{\Gamma \vdash e_p : \tau_p}{\Gamma \vdash \texttt{advance}\ e_p : ()}$$

T-APP
$$\frac{\Gamma \vdash e : \tau_1 \to \tau_2 \setminus C \qquad \Gamma \vdash e' : \tau_1 \setminus C'}{\Gamma \vdash e\ e' : \tau_2 \setminus C \cup C'}$$

T-ARROW
$$\frac{\Gamma,\ x : \tau_1,\ p : \tau_p,\ k : \tau_2 \to (),\ h : \tau_3 \to () \vdash e : () \setminus C}{\Gamma \vdash \lambda x.\ \lambda p.\ \lambda k.\ \lambda h.\ e : \tau_1 \to \tau_p \to (\tau_2 \to ()) \to (\tau_3 \to ()) \to () \setminus C}$$

T-ARROW-APP
$$\frac{\begin{array}{c} \Gamma \vdash e_a : \tau_1 \to \tau_p \to (\tau_2 \to ()) \to (\tau_3 \to ()) \to () \setminus C \\ \Gamma \vdash e_p : \tau_p \qquad \Gamma \vdash e : \tau_1 \setminus C_1 \\ \Gamma \vdash e_k : \tau_2 \to () \setminus C_2 \qquad \Gamma \vdash e_h : \tau_3 \to () \setminus C_3 \end{array}}{\Gamma \vdash e_a \bullet (e,\ e_p,\ e_k,\ e_h) : () \setminus C \cup C_1 \cup C_2 \cup C_3}$$

T-ASYNC
$$\frac{\begin{array}{c} \Gamma \vdash e_e : \langle Succ : \tau_1, Fail : \tau_2 \rangle \\ \Gamma \vdash e_p : \tau_p \qquad \Gamma \vdash e_k : \tau_1 \to () \setminus C_1 \qquad \Gamma \vdash e_h : \tau_2 \to () \setminus C_2 \end{array}}{\Gamma \vdash \texttt{async}\ \ e_e\ e_p\ e_k\ e_h : () \setminus C_1 \cup C_2}$$

Figure 19. Typing rules for expressions in abstract syntax.

in the same tree but not on the same path as e_p. Note that while ϵ can represent the root of any tree, $P_i^1 :: \epsilon$ and $P_j^2 :: \epsilon$ are not in the same tree if $i \neq j$.

Application of abstract arrows are represented by expression of the form $e_a \bullet (e,\ e_p,\ e_k,\ e_h)$. Asynchronous arrows are translated to abstract arrows with bodies of the form `async` $e_e\ e_p\ (\lambda y.\ e)\ (\lambda z.\ e')$, where e_e ranges over time or Ajax event objects. The meta-variable e_l ranges over tagged unions of `loop`(e) and `halt`(e) that are used to determine whether repeated arrows should loop or halt.

C.2 Translation to Abstract Syntax

Figure 17 defines the translation from concrete arrows to expressions in continuation passing style. For simplicity, the translation rules for the n-ary combinators `seq`, `product`, and `any` are defined as binary combinators. The extension of these translation rules to support $n \geq 2$ arrows is trivial (but notationally dense). We omit the translation of `elem`, `split`, and `nth` arrows as they can be translated from simple lifted functions. To further reduce clutter, the semantics does not include `noemit` combinator and `event` constructor.

The arrow `lift`(f) is translated to an expression that applies the continuation function k to the result produced by invoking f. We omit the exceptions thrown by lifted functions since the main focus is on the resolution and rejection of asynchronous events. The `ajax` and `delay` arrows are translated to expressions that register callback functions to AJAX and time events, respectively. The progress list p is advanced in the callback functions of these arrows in order to create an observable yield point (discussed further in Section C.3).

The combinator `repeat`(a) is translated to a recursive abstract arrow which will either re-invoke the arrow with the same progress list and continuation, or call the continuation with the result of a (dependent on the result of a). The progress list p is advanced in the continuation of a in order to create an observable yield point. The translation rules for the `seq` and `product` combinators are straightforward. The combinator `any`(a_1, a_2) is translated to two abstract arrows $[\![a_1]\!]$ and $[\![a_2]\!]$ with progress lists $P_i^1 :: p$ and $P_i^2 :: p$, respectively. As we will show next, the two progress lists ensure that if the execution of a_1 makes progress, then a_2 is canceled (and the opposite).

The term a.`run`() is translated to an expression that applies $[\![a]\!]$ to a dummy callback function and returns unit [2].

C.3 Operational Semantics

Figure 18 defines the operational semantics of abstract arrows, where the context Δ maps event objects to triples of progress lists, event handlers, and exception handlers.

We distinguish between normal JavaScript function application (E-Host-App) and arrow application (E-Arrow-App) by inserting runtime type checks in the former to ensure that the input and output of a call to f are consistent with the declared type of f (which is supplied by the user). A possible runtime error is when the result of a call to f is a value of an unexpected type, but this can only occur if f is incorrectly annotated.

Rule (E-Async) adds a mapping from the event object v_e to a triple of a progress list, a callback handler, and an exception handler $(v_p,\ \lambda y.\ e,\ \lambda z.\ e')$, to Δ. Rule (E-Event-Succ) invokes the handler of an event object v_e if it *occurs* with a success, where $Resp(.)$ is a function that returns the response to v_e, which is either $Succ(v)$ (success with result v) or $Fail(v)$ (failure with exception v). Rule (E-Event-Fail) invokes the exception handler of v_e if it *occurs* with a failure. A time event *occurs* once a particular number of millisecond pass since the event was registered and an AJAX event *occurs* once the corresponding AJAX request is answered. The two event rules simulate the JavaScript event loop. Both event rules insert runtime checks to ensure that the responses to events, (e.g. AJAX requests), have the correct types. This type of errors can only occur if event arrows (e.g. AJAX arrows) are incorrectly annotated.

By Rule (E-Advance), `advance`(v_p) recursively removes event handlers with the progress lists that are in the same tree as v_p but not on the same path. This is used by the `any` combinator so that once an arrow passes its first yield point all other pending arrows within the combinator are canceled. Notice that each arrow is associated with a progress list and the arrow `any`(a_1, a_2) extends its progress list v_p with a fresh progress object P_i^j and passes them to a_j where $j \in \{1, 2\}$. If an arrow with a progress list that includes P_i^j makes progress, then the arrows with progress lists that include P_i^k are canceled through `advance`$(P_i^j :: v_p)$, where $k \neq j$.

The standard reduction rules for tuple, tuple projection, sequence, tail call, fix, and case expressions are omitted.

C.4 Properties

To show that the arrow type system is sound, we first establish that the translation of concrete arrows preserves typing (Theorem C.1). The typing rules for arrows in abstract syntax are shown in Figure 19, where the rules for tuple, tuple projection, sequence, fix expression, continuation, function value, and case expressions are omitted. Most of the rules are not surprising. The typing rules have the implicit assumption that all type variables are fresh and if $\Gamma \vdash e : \tau \setminus C$, then C is consistent. The proof of this theorem is straightforward and omitted.

Theorem C.1. If $a : \tau_1 \rightsquigarrow \tau_2 \setminus (C, E)$, then $\emptyset \vdash [\![a]\!] : \tau_1 \to \tau_p \to (\tau_2 \to ()) \to (\tau_3 \to ()) \to () \setminus C \cup \{\tau \leq \tau_3 \mid \tau \in E\}$, where τ_p is the type of all e_p.

Next, we show that the execution of a well-typed arrow will not get stuck in Theorem C.2, which states if an arrow a is well-typed and takes no input, then its execution will reduce to unit and all the event handlers generated by a will run to completion as well. We make the assumption that all events will occur eventually, and the event handlers added to Δ are eventually invoked.

Theorem C.2. If $a : () \rightsquigarrow \tau \setminus (C, \emptyset)$, then either $\epsilon, [\![a.\texttt{run}()]\!] \to^* \epsilon, ()$ or it gets stuck because of Rule (T-Host-App), (E-Event-Succ), or (E-Event-Fail).

By Theorem C.1, and because a is well-typed, it is clear that $[\![a.\texttt{run}()]\!]$ is well-typed. It is straightforward to show that well-typed expressions can make progress or get stuck because of Rule (E-Host-App), (E-Event-Succ), or (E-Event-Fail).

To show that subject reduction preserves typing, we concentrate on these three rules. Rule (E-Host-App) states $\Delta, f\ v$ reduces to Δ, v' if $f\ v$ reduces to v' and the constraint set closure of the type of v' is a subset of the constraint set closure of the type of $f\ v$. Suppose $\epsilon, [\![a.\texttt{run}()]\!] \to^* \Delta, E[f\ v]$ and $\Delta, f\ v \to \Delta, v'$, where $E[\cdot]$ is an evaluation context. If C and C' are the constraint sets of the types of $E[f\ v]$ and $E[v']$, respectively, then $\texttt{Cl}(C') \subseteq \texttt{Cl}(C)$. Since $E[f\ v]$ is well-typed and C is consistent, C' is consistent and $E[v']$ is well-typed. Rule (E-Async) and (T-Async) ensure that $\forall v_e \mapsto (v_p,\ \lambda y.e,\ \lambda z.e') \in \Delta$, if $\emptyset \vdash v_e : \langle succ : \tau_1, fail : \tau_2 \rangle$, then $\emptyset \vdash \lambda y.e : \tau_1 \to () \setminus C_1$ and $\emptyset \vdash \lambda z.e' : \tau_2 \to () \setminus C_2$, and $C_1 \cup C_2$ is consistent. If the reduction is by Rule (E-Event-Succ), where $\Delta, () \to \Delta \setminus \{v_e \mapsto (v_p,\ \lambda y.e,\ \_)\},\ [v/y]e$, then $Resp(v_e) = Succ(v), \emptyset \vdash v : \tau_1$, and $\emptyset \vdash v_e : \langle Succ : \tau_1, Fail : \_ \rangle$. Thus, $\emptyset \vdash [v/y]e : () \setminus C_1$ and C_1 is consistent. The reduction for Rule (E-Event-Fail) is similar.

[2] In practice, we return a progress object from a.`run`() so that the user is able to cancel the event handlers generated by the execution of a.