Elsevier Editorial System(tm) for Science of Computer Programming

Manuscript Draft

Manuscript Number:

Title: Typing and Semantics of Asynchronous Arrows in JavaScript

Article Type: Research Paper

Keywords: JavaScript; Asynchronous Programming; Type Inference; Type Checking; Semantics

Corresponding Author: Dr. Tian Zhao, Ph.D.

Corresponding Author's Institution: University of Wisconsin -- Milwaukee

First Author: Eric Fritz, MS

Order of Authors: Eric Fritz, MS; Tian Zhao, Ph.D.

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.

Highlights (for review)

- an arrows-based DSL in JavaScript which encodes semantics similar to ES6 Promises
- an optional type system to support type-directed composition of asynchronous arrows
- a formal semantics of arrow and its relation to JavaScript's single-threaded event loop

Typing and Semantics of Asynchronous Arrows in JavaScript

Eric Fritz, Tian Zhao

University of Wisconsin – Milwaukee

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.

Keywords: Design Tools and Techniques, Software Libraries, Testing and Debugging, Error handling and recovery

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 [1] 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* [2], which is a generalization of monads [3].

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,
2. an optional type system to aid developers with type-directed composition of asynchronous arrows, and
3. a formal semantics that describes the implementation of asynchronous arrow and its relation to JavaScript's single-threaded event loop.

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 describes formal semantics of asynchronous arrows. Section 6 discusses a soundness proof for arrow type inference. Section 7 discusses the runtime cost and the development overhead of our library. Section 8 presents related work and Section 9 concludes. Our arrows library, the type-checker, and some sample applications are freely available[1].

2. Example

To illustrate the utility of our type inference tool, consider the example in Figure 1 which implements an autocomplete feature for a text field. In this

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

example, Ajax requests which include the user's input are made to a remote server. The response from the server contains results matching the user's input and is displayed on the page. Ajax requests are triggered on keyup events so that the displayed results update as the user types.

In addition to the basic autocomplete functionality, this example also implements two small optimizations, as follows.

1. Ajax responses are cached locally so the same request is not made twice. This can easily be modified to include an expiry so that cached results do not become too stale.

2. The number of Ajax requests are limited. A request is only launched 500ms after the user's last keystroke in order to prevent spurious requests, the results of which will simply be discarded as new information is available by the time it is ready.

The details of the arrow methods are explained in Section 3. For now, it should be sufficient to understand that `LiftedArrow` converts a function into an arrow, `seq` chains two arrows in sequence, `fix` allows an arrow to reference itself recursively, `any` executes two arrows in parallel and cancels the execution of the *slower* of the two, `catch` executes the second arrow when an exception occurs in the first, and `on` blocks until an event to occur on an element. We encourage readers to reference Section 3 in order to understand the complete composition (shown in lines 29-40 of Figure 1). The *derived* `remember` arrow discards the output of the arrow it wraps and returns its input. This arrow is discussed in Section 7. 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.

The arrow `ajaxOrCached` (composed in lines 29-31 of Figure 1) abstracts the machinery of a cached Ajax request. This arrow has the inferred type of $\mathit{String} \rightsquigarrow [\mathit{String}]$, accepting a string key and returning an array of strings. The result comes from either the cache or from the remote server. First, the lifted

```
var cache = {};
let lookup = new LiftedArrow(key => {
  /* @arrow :: String ~> α \ ({}, { String }) */
  if (!(key in cache)) throw key;
  return cache[key];
});

let store = new LiftedArrow((key, value) => {
  /* @arrow :: (α, β) ~> ⊤ */
  cache[key] = value;
});

let getVal = new LiftedArrow((elem, event) =>
  /* @arrow :: (\DOMElem, \DOMEvent) ~> String */
  $(elem).val()
);

let handle = new LiftedArrow(results => {
  /* @arrow :: [String] ~> ⊤ */
  $('#results').empty().append(results.map(t => $('<li />').text(t)));
});

let ajax = new AjaxArrow(q => {
  /* @conf :: String ~> {url: String}
     @resp :: [String] */
  return { 'url': host + '?q=' + q + '&limit=100' };
});

let ajaxOrCached = Arrow.catch(lookup,
  Arrow.seq(ajax.carry(), store.remember()).nth(2)
);

Arrow.fix(a => Arrow.seq([
  new ElemArrow('#input'),
  Arrow.on('keyup', Arrow.any([
    Arrow.seq([getVal, new Delay(500), ajaxOrCached, handle]),
    a
  ])),
  a
])).run();
```

Figure 1: An example arrow composition which fetches results from a remote server as the user types.

function `lookup` is invoked, which will either return a value from the cache on cache-hit or throw the key on cache-miss. If the function throws, the exception value is caught and fed into the `ajax` arrow, then stored in the cache. The execution of the arrow halts, yielding the cached value.

Notice that `ajax` returns an array of *String* objects, but the `store` function requires both a key and a value as input. Sequencing them directly would result in a (rightful) type-clash. The non-primitive `carry` combinator converts an arrow with type $\alpha \rightsquigarrow \beta$ into an arrow with type $\alpha \rightsquigarrow (\alpha,\ \beta)$, piping the input of the arrow to the output.

Omitting the `carry` combinator would results in the error

results.map is not a function

from within the `handle` function during runtime if type-checking was not enabled. This occurs because the result of the Ajax request is unpacked into the parameters of the `store` function such that `key` is bound to the first result in the list and `value` is bound to the second. If type-checking was enabled, the sequencing of `ajax` and `store` instead fails with the following type clash at composition time, giving the developer immediate feedback that the arrows are ill-composed.

Cannot seq arrow: Inconsistent constraints $\{[\text{String}] \leq (\text{String},\ \alpha)\}$

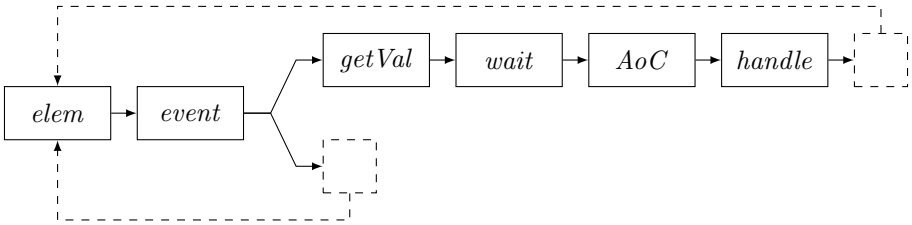


Figure 2: Simplified arrow composition topology of the code in Figure 1. The dashed boxes represent a reference to another arrow.

The arrow composed in lines 33-40 of Figure 1 is composed with a recursive reference to itself. This arrow is shown graphically in Figure 2 and has the

inferred type of $\top \rightsquigarrow \alpha \setminus (\emptyset, \{AjaxError\})$, meaning that the arrow accepts *any* value (as it is not used in a meaningful way within the arrow), produces an unbounded value due to an infinite execution cycle, and may throw a value of type *AjaxError*.

Once a *keyup* event occurs on the element with a particular identifier, the arrow begins to execute two arrows in parallel (a *delay* arrow and the recursive reference a), waiting for one to *progress*. The delay arrow waits 500 milliseconds before continuing. The arrow a blocks until another keyup event occurs. If the interval passes without a keyup event, then the recursive invocation of a is canceled and the delay path continues to execute. If the user triggers a keyup event before the interval passes, then the delay path is canceled and the arrow effectively *restarts*. A cached Ajax request (described above) is performed and the results are passed to the `handle` function, which inserts them into the web page. After `handle` executes, the arrow is executed recursively, waiting for the user's next keyup event.

While the arrow composition is recursive, it requires a constant amount of stack space. This is because each recursive call begins with an arrow that registers a continuation function with an event queue. When the event fires, the continuation is pulled off the queue and invoked, but does not retain the stack context in which the continuation was created.

In the following section, we describe the full set of *core* arrows which are used to create complex applications, as seen above.

3. Arrows

An arrow is a composable, opaque unit of execution which, in this context, runs in an asynchronous manner. An arrow may receive an argument, but may only receive it 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 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 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 un-registers all of its active event handlers so that it is never notified to resume execution when an external event occurs.

Definition 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 [1] and ES6 Promises. We have, however, made a few major interface changes which are discussed in Section 8.

3.1. Constructors

We provide seven arrow constructors, detailed below.

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. A concrete example is given in Figure 3.

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

Figure 3: An example of a function *lifted* into an arrow.

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. We assume lifted functions execute in a synchronous manner.

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

DOM Element. The element arrow, denoted `domelem`(*selector*), produces a jQuery object matching the given selector. A jQuery object denotes a (possibly-empty) *set* of objects, so that each invoked method is delegated to the elements of the set.

DOM Event. The event arrow, denoted `domevent`(*name*), takes a DOM element as input and produces an event value specific to the action denoted by *name after that event occurs on the given element.* This arrow is asynchronous. Concretely, the arrow `domevent`(*click*) arrow takes a jQuery object as input and returns a *click* event once *any* of the elements in the input element set are clicked by the user.

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.

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

Figure 4: An example of an arrow which asynchronously fetches data form a remote server.

If type-checking is enabled, it is expected that c is annotated with the input and output 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. A concrete example is given in Figure 4.

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 5: Dataflow diagrams for `split` and `nth` arrows.

Note that the `domelem` 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 five arrow combinators, detailed below. Async points are represented in dataflow diagrams as double-slashed lines. The `noemit` combinator transforms a single arrow. The `try` combinator transforms three arrows. The remaining four combinators transform a set of $n \geq 1$ arrows.

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.

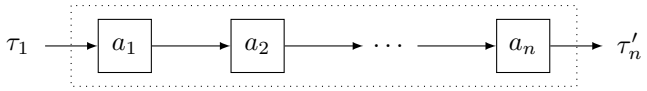


Figure 6: Dataflow diagram for the `seq` combinator.

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.

This combinator generalizes the binary combinator

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

in the arrow calculus [2].

All. The all combinator, denoted `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.

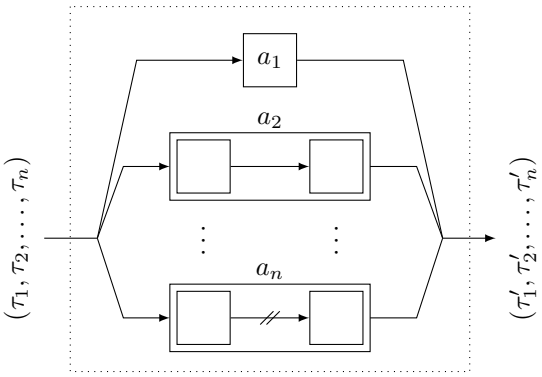


Figure 7: Dataflow diagram for the `all` combinator over arrows a_1 through a_n. The argument arrows can be simple or a complex result of other combinators (a_1 and a_2, respectively), and can be synchronous or asynchronous (a_2 and a_n, respectively).

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 [2] using this combinator and an *identity* arrow:

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

Try. The try combinator, denoted $\texttt{try}(a, a_s, a_f)$, 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.

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

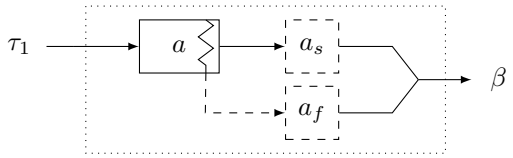


Figure 8: Dataflow diagram for the `try` combinator.

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.

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

The `try` combinator can also support a (restricted) encoding of conditional execution. The protected arrow can return a value to signify that the *true* (success) path should be executed, and throw a value to signify that the *false* (failure) path should be executed. An example of this was shown in Section 2 by the `lookup` arrow, which returns a value on cache-hit and throws a value on cache-miss. Encoding conditional execution with `try` does not work in all circumstances. Because the failure arrow must be able to accept *any* exception thrown by the protected arrow, this pattern will not work when the protected arrow throws exceptions unrelated to control flow.

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 input value v.

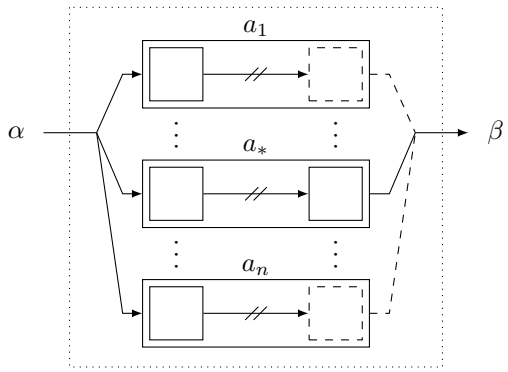


Figure 9: Dataflow diagram for the `any` combinator. Notice that all arrows have an async point.

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 first async point of a_n. Once 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 handler for a timer event and immediately yield, where $game_1$ begins to execute. If $game_1$ completes, then the timer is canceled. If the timer completes, then the sequence is canceled. The `noemit` combinator additionally suppresses the progress event emitted by $game_1$ so that it does not cancel timer before $game_1$ completes.

3.3. Recursion and Repeating

The combinators presented in the previous section are all linear, and composed arrows always form a directed acyclic graph. Unfortunately, many user

interaction patterns for which arrows would be useful require some sort of repetition. Such patterns include, but are not limited to, continuously responding to an event and using bounded repetition in the context of animation.

To support such uses, we allow arrows to be defined *recursively*. This allows a restricted type of cycle to be introduced to the composition graph. A recursive arrow is constructed by using a fixed-point primitive, `fix`, defined as

$$\texttt{fix}(\omega \texttt{ => } e)$$

where ω is a reference to the arrow being defined, and e is an expression which constructs an arrow referencing ω.

In practice, this primitive is implemented by constructing an empty *proxy arrow* a, passing a to the function ω `=>` e which results in the arrow a', then replacing the proxy arrow a by reference with a'. This method allows a logically *infinite* arrow to be constructed in constant time and space.

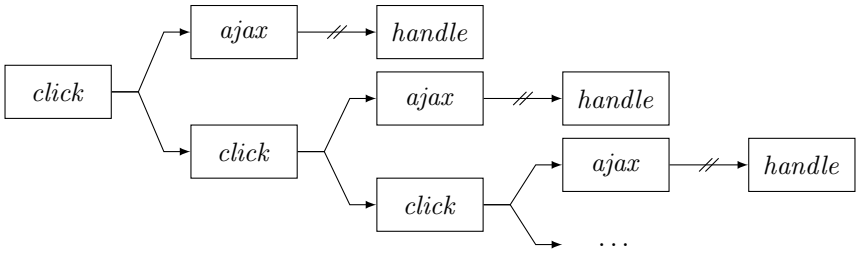


Figure 10: A recursive arrow composition.

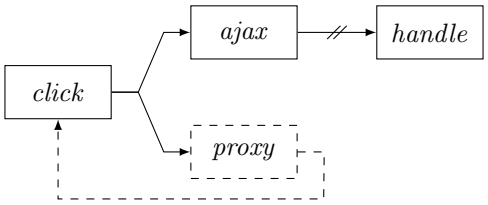


Figure 11: A recursive arrow composition (with a proxy arrow).

Figure 10 shows a snippet of an arrow which cannot be constructed without recursion. In this composition, a *click* event triggers an *ajax* request to fire, and

the result is passed to *handle*. If a subsequent *click* event occurs before the *ajax* request has completed, then the request is canceled and a new *ajax* request is fired. This arrow can be easily expressed as the following recursive arrow.

$$\texttt{fix}(\omega \texttt{ => } \texttt{seq}(\mathit{click},\ \texttt{or}(\texttt{seq}(\mathit{ajax},\ \mathit{handle}),\ \omega)))$$

The resulting arrow is shown in Figure 11. When execution hits the proxy arrow, it simply executes its reference with the proper arguments.

Our previous work included an additional combinator, `repeat`, for a restricted form of executing an a arrow *at least* once [4]. The combinator executes a, then uses its output to determine if it should execute the arrow again. If the arrow is re-executed, its output is fed back into itself. Otherwise, the arrow halts and yields its most recent value. The combinator expects the result of a to be tagged union of the form $\langle \mathit{loop} : \tau_1,\ \mathit{halt} : \tau_2 \rangle$, implemented as a simple object type with a tag and value field. This enables the decision to repeat to be made by reading the tag value.

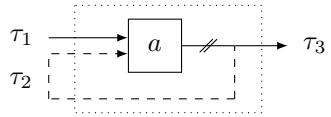


Figure 12: Dataflow diagram for the `repeat` combinator.

The 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.

With the inclusion of `fix`, the `repeat` combinator can be encoded as

$$\texttt{repeat}(a) \ \equiv\ \texttt{fix}(\omega \texttt{ => } \texttt{seq}(a,\ \texttt{delay}(0),\ \texttt{try}(\mathit{repeatTail},\ \omega,\ \mathit{id})))$$

where the arrow *repeatTail* takes the result of a and either returns τ_1 or throws τ_2. *repeatTail* is implemented as a lifted function, show in Figure 13.

The `try` combinator will pass the value τ_1 recursively to ω, or pass the thrown value τ_2 to the identity arrow. Notice that any invocation of a is **not**

```
let repeatTail = new LiftedArrow(x => {
  /** @arrow :: <loop: α, halt: β> ⇝ α \ ({}, {β}) */
  if (x.hasTag('loop')) {
    return x.value();
  } else {
    throw x.value();
  }
});
```

Figure 13: The implementation of the arrow *repeatTail*.

protected by a `try` combinator, so any exceptional value produced by a will be properly thrown from the recursive arrow. The async point is added by sequencing a zero-delay time event after each invocation of a.

3.4. 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 error callback h, we give the CPS encodings for the `lift` constructor and the `try` combinator in Figure 14 and Figure 15, respectively.

To demonstrate the use of the progress object p, we give the CPS encoding for the `delay` constructor and `any` combinator in Figure 16 and Figure 17. The `any` combinator creates a fresh progress object for each of its children. An observer is registered to each progress object to be notified when a progress event with respect to that object is fired. When one progress object emits a progress event, its sibling arrows are canceled. For brevity, we show a the binary version of the `any` combinator. In practice, this combinator can wrap $n \geq 1$ arrows.

```
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 14: 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 15: Encoding for `try`(a, a_s, a_f).

```
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 16: Encoding for `delay`(*duration*).

```
call(x, p, k, h) {
  const p1 = new Progress();
  const p2 = new Progress();

  // Canceling parent progress cancels children as well
  const cancel = () => { p1.cancel(); p2.cancel(); }
  p.addCanceler(cancel);

  // When pi makes progress, cancel pj and emit an event
  p1.addObserver(() => { p2.cancel(); p.advance(cancel) });
  p2.addObserver(() => { p1.cancel(); p.advance(cancel) });

  // Execute arrows in order
  a1.call(x, p1, k, h);
  a2.call(x, p2, k, h);
}
```

Figure 17: Encoding for $\texttt{any}(a_1,\ a_2)$.

The `noemit` combinator creates a fresh progress object which does not emit the progress events of the arrow it wraps, but emits a single progress event on completion. The encoding for `noemit` is omitted.

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 JavaScript objects which facilitate DOM events (e.g. *DomElem*, *DomEvent*), we define the type of *primitive* values, denoted b, as follows.

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

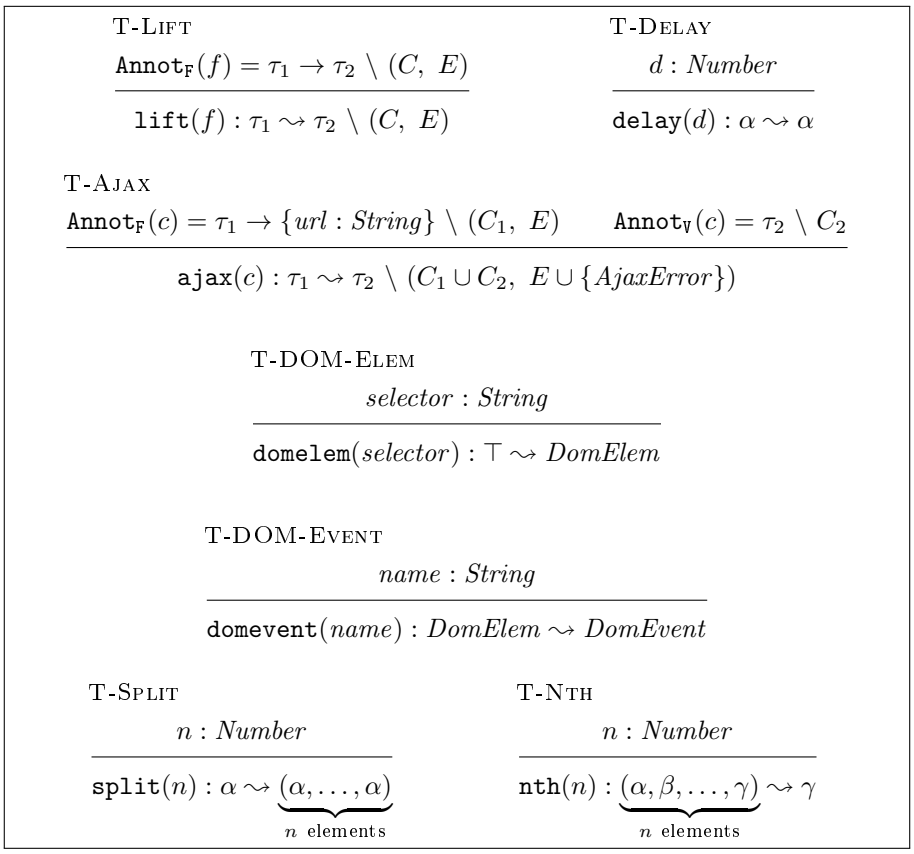


Figure 18: Typing rules for arrow constructors.

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.

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

The *top* (*any possible*) type is represented by $\top$. We assign $\top$ type to the Javascript value `undefined` and `null` and use it as the return type of any JavaScript function that does not return values. The type $\top$ is also commonly

T-Seq

$$\frac{\Gamma \vdash a_i : \tau_i \rightsquigarrow \tau_i' \setminus (C_i,\ E_i) \qquad C' = \cup_{i=2}^{n} \{\tau_{i-1}' \leq \tau_i\}}{\Gamma \vdash \texttt{seq}(a_1, \ldots, a_n) : \tau_1 \rightsquigarrow \tau_n' \setminus (C' \cup \bigcup C_i,\ \bigcup E_i)}$$

T-All

$$\frac{\Gamma \vdash a_i : \tau_i \rightsquigarrow \tau_i' \setminus (C_i,\ E_i)}{\Gamma \vdash \texttt{all}(a_1, \ldots, a_n) : (\tau_1, \ldots, \tau_n) \rightsquigarrow (\tau_1', \ldots, \tau_n') \setminus (\bigcup C_i,\ \bigcup E_i)}$$

T-Try

$$\frac{\Gamma \vdash a_i : \tau_i \rightsquigarrow \tau_i' \setminus (C_i,\ E_i) \qquad C' = \{\tau_1' \leq \tau_2,\ \tau_2' \leq \beta,\ \tau_3' \leq \beta\} \cup \{\tau \leq \tau_3 \mid \tau \in E_1\}}{\Gamma \vdash \texttt{try}(a_1, a_2, a_3) : \tau_1 \rightsquigarrow \beta \setminus (C' \cup \bigcup C_i,\ E_2 \cup E_3)}$$

T-Any

$$\frac{\Gamma \vdash a_i : \tau_i \rightsquigarrow \tau_i' \setminus (C_i,\ E_i) \qquad C_i' = \{\alpha \leq \tau_i,\ \tau_i' \leq \beta\}}{\Gamma \vdash \texttt{any}(a_1, \ldots, a_n) : \alpha \rightsquigarrow \beta \setminus (\bigcup C_i' \cup \bigcup C_i,\ \bigcup E_i)}$$

T-Omega

$$\frac{(\omega : \tau_1 \rightsquigarrow \tau_2) \in \Gamma}{\Gamma \vdash \omega : \tau_1 \rightsquigarrow \tau_2}$$

T-NoEmit

$$\frac{\Gamma \vdash a : \widetilde{\tau}}{\Gamma \vdash \texttt{noemit}(a) : \widetilde{\tau}}$$

T-Fix

$$\frac{\Gamma,\ \omega : \alpha \rightsquigarrow \beta \vdash a : \tau_1 \rightsquigarrow \tau_2 \setminus (C,\ E)}{\Gamma \vdash \texttt{fix}(\omega \texttt{ => } a) : \alpha \rightsquigarrow \beta \setminus (C \cup \{\alpha \leq \tau_1,\ \tau_2 \leq \beta\},\ E)}$$

Figure 19: Typing rules for arrow combinators.

used as an upper bound for variables which have incompatible types (such as numbers and booleans).

A tagged union of type $\langle \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \rangle$ holds a single value of type τ_i, which is accessible by querying the associated tag ℓ_i. These values are represented by a simple JavaScript object with a tag and a value field. In particular, the tagged union $\langle \mathit{loop} : \tau_1,\ \mathit{halt} : \tau_2 \rangle$ is used to support the encoding of the `repeat` combinator, as discussed in Section 3.3. 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.

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 as

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

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.

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. Section 4.3 outlines an algorithm for determining whether a constraint set is consistent. In brief, the algorithm rejects constraint sets whose closure contains obvious subtyping violations.

The constrained arrow type is similar to the constrained type $\tau \setminus C$ introduced by Eifrig et al. [5], 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 18 and Figure 19, 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 function 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: a constrained function type describing the input and output types of c, and a constrained value type describing the response from the remote server. We assume the existence of the implicit functions $\texttt{Annot}_{\texttt{F}}(f)$ and $\texttt{Annot}_{\texttt{V}}(f)$ which reads the function or value annotations (respectively) from the function f and produces a unique instantiation of the type it describes.

Rule (T-Nth) shows how the $\texttt{nth}(n)$ 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.

The Rule (T-Fix) is discussed in detail in Section 4.4.

4.3. Consistency

$$\textsc{Cls-Trans}\ \frac{\{\tau_1 \leq \tau_2, \tau_2 \leq \tau_3\} \subseteq C}{\tau_1 \leq \tau_3 \in C} \qquad \textsc{Cls-Array}\ \frac{[\tau] \leq [\tau'] \in C}{\tau \leq \tau' \in C} \qquad \textsc{Cls-Union}\ \frac{\langle \ell_i : \tau_i \rangle^{i \in 1..k} \leq \langle \ell_i : \tau'_i \rangle^{i \in 1..n} \in C}{\{\tau_i \leq \tau'_i\}^{i \in 1..k} \subseteq C}$$

$$\textsc{Cls-Tuple}\ \frac{(\tau_i)^{i \in 1..n} \leq (\tau'_i)^{i \in 1..k} \in C}{\{\tau_i \leq \tau'_i\}^{i \in 1..k} \subseteq C} \qquad \textsc{Cls-Record}\ \frac{\{\ell_i : \tau_i\}^{i \in 1..n} \leq \{\ell_i : \tau'_i\}^{i \in 1..k} \in C}{\{\tau_i \leq \tau'_i\}^{i \in 1..k} \subseteq C}$$

Figure 20: Constraint set closure rules, where $k \leq n$.

In this section, we present the definition of constraint set consistency. An

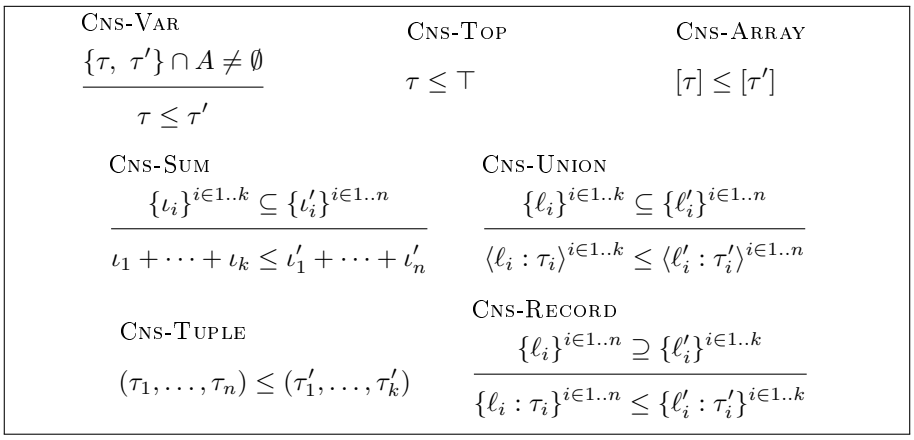


Figure 21: Constraint set consistency rules, where $k \leq n$.

arrow is *inconsistent* if a constraint present an *immediate inconsistency*. For example, $\mathit{String} \leq \mathit{Number}$ is immediately inconsistent, as is $(\tau, \tau) \leq \{\ell : \tau\}$.

Definition 1 (Closed)**.** A set of constraints C is *closed* if it satisfies the closure rules given in Figure 20. We refer to the closure of C as $\mathit{closure}(C)$.

Definition 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 21.

Rule (Cls-Trans) ensures that subtype constraints are transitive. For example, $C = \{\mathit{String} \leq \alpha,\ \alpha \leq \mathit{Number}\}$ contains only consistent constraints. However, $\mathit{String} \leq \mathit{Number} \in \mathit{closure}(C)$ and the constraint set is therefore considered inconsistent. This occurs because there is no type for α which satisfies its bounds.

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). The presence of a top type allows an arrow consuming no *useful* value to be composed with any other 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. 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 $T \subseteq T'$.

Rules (Cns-Union), (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. $C = \{\alpha \leq \mathit{String},\ \alpha \leq \mathit{Number}\}$, where no lower bound of both *String* and *Number* exists). This case is handled by type simplification, discussed in Section 4.5.

4.4. Typing Recursive Arrows

Rule (T-Fix) leads to an algorithmic approach to typing recursive arrows that we use in practice. As discussed in Section 3.3, an empty *proxy arrow* takes the place of recursive reference during construction, and is later updated by reference. The proxy arrow is given the permissive type $\alpha \rightsquigarrow \beta$, which can compose legally with any other arrow. Once the arrow is constructed, we refine the type of the proxy arrow to match the type of the derived arrow. The constraints $\{\alpha \leq \tau_1,\ \tau_2 \leq \beta\}$ ensure that an arrow of type $\tau_1 \rightsquigarrow \tau_2$ can be used whenever $\alpha \rightsquigarrow \beta$ is expected.

4.5. Type Simplification

In this section, we discuss a *type simplification* technique for arrow types. Details of simplification is given in the appendix. 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 $\mathit{DomEvent} \rightsquigarrow \top$.

$$\texttt{seq}(\texttt{split}(2),\ \texttt{all}(\mathit{id},\ \texttt{seq}(\texttt{domevent}(\mathit{click}),\ a),\ \texttt{nth}(1))$$

The resulting arrow takes as input an *DomElem* value, waits until a *click* events occurs on that value, invokes the arrow a with the resulting event, and then yields the original *DomElem* 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 as follows.

$$\alpha \rightsquigarrow \delta \setminus (\{\ \mathit{DomEvent} \leq \mathit{DomEvent},\ (\alpha,\ \alpha) \leq (\mathit{DomElem},\ \beta),$$
$$(\top,\ \beta) \leq (\gamma,\ \delta),\ \alpha \leq \mathit{DomElem},\ \alpha \leq \beta,\ \top \leq \gamma,$$
$$\beta \leq \delta,\ \alpha \leq \delta\ \},\ \emptyset)$$

The first constraint, $\mathit{DomEvent} \leq \mathit{DomEvent}$, is introduced by the inner `seq`. The next two constraints $(\alpha,\ \alpha) \leq (\mathit{DomElem},\ \beta)$ and $(\top,\ \beta) \leq (\gamma,\ \delta)$, 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 4.3. After type simplification we are given the following, which is much smaller and more easily understandable.

$$\alpha \rightsquigarrow \delta \setminus (\{\ \alpha \leq \delta,\ \alpha \leq \mathit{DomElem}\ \},\ \emptyset)$$

The *Memory* application mentioned in Section 7 creates an arrow with $1,414$ distinct constraints involving 56 type variables with type simplification disabled, and **no** constraints after minimization.

Simplifying Recursive Arrows. Type simplification presents a problem with typing recursive arrows that is not present in linearly composed arrows. Linearly

composed arrows have the full set of constraint information necessary to infer a type at the time the type of the combinator is inferred. This correctly allows an arrow type to be simplified immediately. Recursively composed arrows, on the other hand, add four additional constraints on α and β to the arrow's constraint set *after* composition occurs. It is possible to infer an incorrect type by prematurely simplifying intermediate arrow types.

For example, consider the arrow with the recursive definition

$$\texttt{fix}(\omega \texttt{ => } \texttt{seq}(a,\ \omega))$$

where a has type $\iota_1 \rightsquigarrow \iota_2$ where ι_1 and ι_2 are named types. The typing of this arrow is shown below. Because $\{\iota_2 \leq \alpha,\ \alpha \leq \iota_1\} \in C$, $\{\iota_2 \leq \iota_1\} \in \mathit{closure}(C)$ and the arrow (correctly) fails to type check.

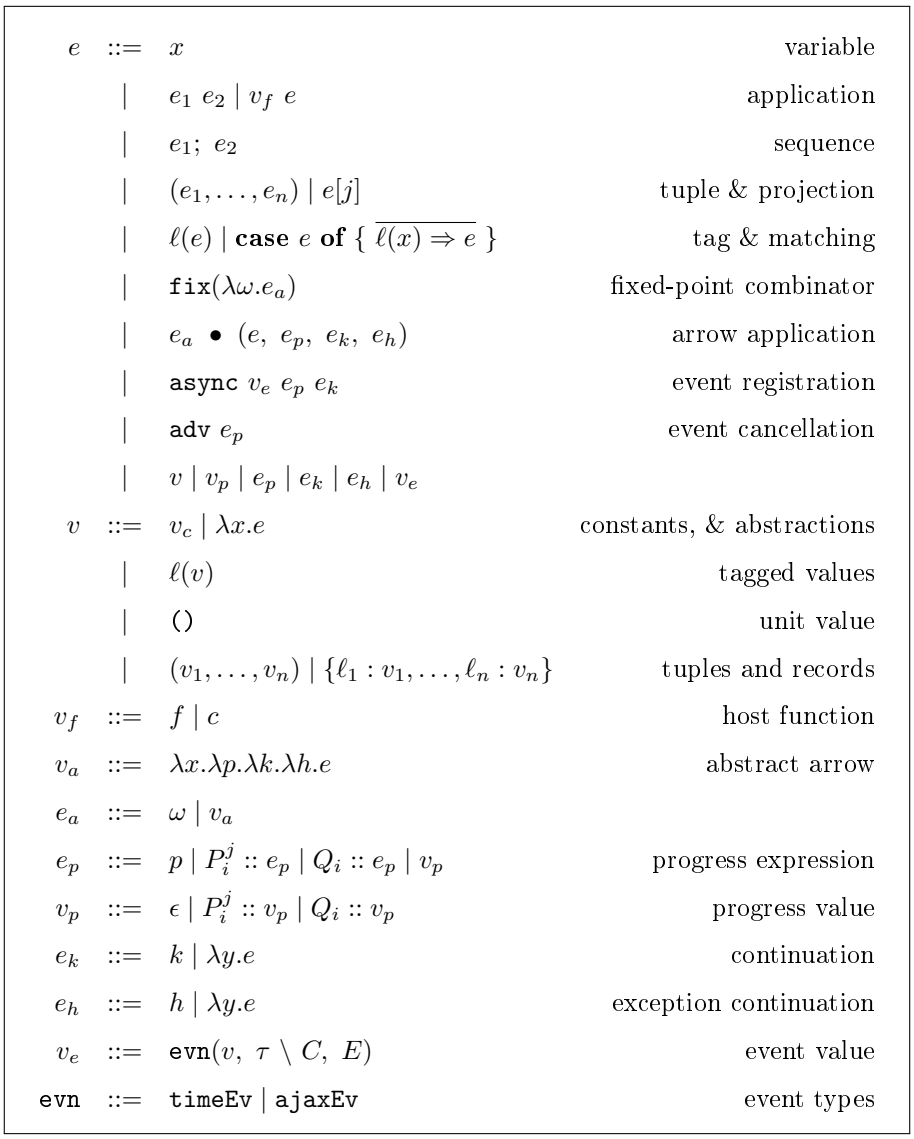
$$\frac{\dfrac{\omega : \alpha \rightsquigarrow \beta \vdash \omega : \alpha \rightsquigarrow \beta \qquad \omega : \alpha \rightsquigarrow \beta \vdash a : \iota_1 \rightsquigarrow \iota_2}{\omega : \alpha \rightsquigarrow \beta \vdash \texttt{seq}(a,\ \omega) : \iota_1 \rightsquigarrow \beta \setminus (\{\iota_2 \leq \alpha\},\ \emptyset)}}{\emptyset \vdash \texttt{fix}(\omega \texttt{ => } \texttt{seq}(a,\ \omega)) : \alpha \rightsquigarrow \beta \setminus (\{\iota_2 \leq \alpha\} \cup \{\alpha \leq \iota_1,\ \beta \leq \beta\},\ \emptyset)}$$

However, if the type of $\texttt{seq}(a,\ \omega)$ is simplified before being used in the type rule for `fix`, then $\texttt{seq}(a,\ \omega)$ has type $\tau_1 \rightsquigarrow \beta$ where all constraints have been simplified away. No such constraint inconsistency occurs in the type for `fix`, and the arrow can be (incorrectly) recursively composed.

In practice, we temporarily disable type simplification of constraints involving α or β while the fixed-point expression is being evaluated, then simplify the type of the entire arrow all at once.

5. Semantics

In this section, we describe the semantics of both the single-threaded event loop and arrow constructors and combinators. We define a simplified calculus, *abstract arrows*, as an extension of lambda calculus in Section 5.1 and a define the operational semantics in Section 5.2. A translation from concrete to abstract arrows in presented in Section 5.3.



e	::=	x	variable
	|	$e_1\ e_2 \mid v_f\ e$	application
	|	$e_1;\ e_2$	sequence
	|	$(e_1, \ldots, e_n) \mid e[j]$	tuple & projection
	|	$\ell(e) \mid \textbf{case}\ e\ \textbf{of}\ \{\ \overline{\ell(x) \Rightarrow e}\ \}$	tag & matching
	|	$\texttt{fix}(\lambda\omega.e_a)$	fixed-point combinator
	|	$e_a \bullet (e,\ e_p,\ e_k,\ e_h)$	arrow application
	|	$\texttt{async}\ v_e\ e_p\ e_k$	event registration
	|	$\texttt{adv}\ e_p$	event cancellation
	|	$v \mid v_p \mid e_p \mid e_k \mid e_h \mid v_e$	
v	::=	$v_c \mid \lambda x.e$	constants, & abstractions
	|	$\ell(v)$	tagged values
	|	$()$	unit value
	|	$(v_1, \ldots, v_n) \mid \{\ell_1 : v_1, \ldots, \ell_n : v_n\}$	tuples and records
v_f	::=	$f \mid c$	host function
v_a	::=	$\lambda x.\lambda p.\lambda k.\lambda h.e$	abstract arrow
e_a	::=	$\omega \mid v_a$	
e_p	::=	$p \mid P_i^j :: e_p \mid Q_i :: e_p \mid v_p$	progress expression
v_p	::=	$\epsilon \mid P_i^j :: v_p \mid Q_i :: v_p$	progress value
e_k	::=	$k \mid \lambda y.e$	continuation
e_h	::=	$h \mid \lambda y.e$	exception continuation
v_e	::=	$\texttt{evn}(v,\ \tau \setminus C,\ E)$	event value
evn	::=	$\texttt{timeEv} \mid \texttt{ajaxEv}$	event types

Figure 22: Abstract syntax.

5.1. Abstract Arrows

Figure 22 defines the complete abstract syntax. We extend lambda calculus with sequencing, tuple values and tuple projection, tagged expression, and tag pattern matching. We include a standard fix-point combinator specifically for

abstract arrows. An abstract arrow, denoted by e_a, has the form

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

where the parameter x denotes the input value of the arrow, the parameter p denotes a *progress list*, the parameter k denotes a continuation function, and parameter h denotes an exception continuation function. Application of abstract arrows is represented by the expression of the form $e_a \bullet (e,\ e_p,\ e_k,\ e_h)$, which is simply sugar for the expression $((((e_a\ e)\ e_p)\ e_k)\ e_h)$. The unit value `()` represents null or undefined values in JavaScript. We use v_c to represent constant values, such as numbers, booleans, and arrays which may be consumed or returned by host functions. For brevity, we do not include typing rules for v_c.

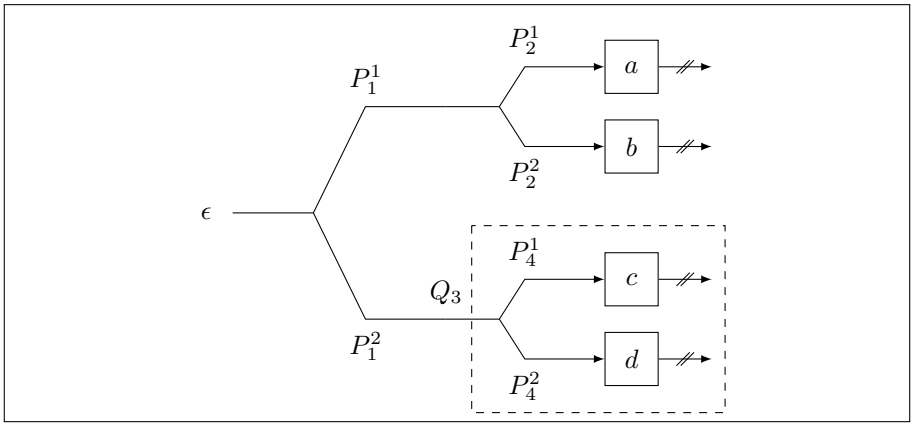


Figure 23: An example progress tree.

A progress list is an ordered sequence of *progress tokens*, which formalizes the progress objects described in Section 3. Progress tokens may be created in pairs, denoted by P_i^1 and P_i^2, or may be created alone, denoted by Q_i. Pairs of progress tokens are used to tag the arms of the `any` combinator so that progress made in one arm can affect the other. A single progress token is used to mark the beginning of the `noemit` combinator.

The progress list is carried by an abstract arrow and grows during its execution - when an arrow begins executing an arm of the `any` combinator, it will prepend a fresh progress token P_i^j to its progress list (and similarly for the

`noemit` combinator). Arrows may share a common prefix of abstract arrows, which allow representing progress lists as a tree of progress tokens. The progress of the arrow

$$\texttt{or}(\texttt{or}(a,\ b),\ \texttt{noemit}(\texttt{or}(c,\ d))$$

is illustrated in Figure 23. Each branch of the `or` combinator gets a unique progress value which is linked to its siblings. The entrance to each `noemit` combinator is signified by the subtree rooted at a progress token Q_i. The progress list at the async point for arrow d is $P_4^2 :: Q_3 :: P_1^2 :: \epsilon$.

When an arrow makes progress within an `any` combinator, the event handlers of the arrows represented by the progress tokens in all other subtrees are canceled. The expression `adv` e_p, in effect, cancels the arrows which are not on the path e_p in the progress subtree rooted at the nearest ancestor of the form ϵ or Q_i. In reference to Figure 23, `adv` P_1^2 will cancel arrows a and b. `adv` P_4^1 will cancel only d; however, `adv` P_2^2 will cancel a, c, and d. Progress objects operating within `noemit` can *be canceled* from outside, but cannot cancel progress objects outside the `noemit` themselves.

Registration of event handlers are represented by the expression

$$\texttt{async}\ v_e\ e_p\ e_k$$

where v_e ranges over time, and Ajax event objects, e_p is the progress list associated with the event (for cancellation), and e_k is a continuation invoked after the event occurs. We explicitly model time and ajax events (corresponding to `delay` and `ajax` arrows), but omit DOM events, as they provide no additional formal interest.

5.2. Operational Semantics

This section defines the operational semantics over expressions of the form

$$\hat{e} ::= e \mid \langle e \rangle$$

where $\langle e \rangle$ denotes an expression which occurs at the top-level. Event callbacks are registered to the event context, Δ, which maps event objects v_e to pairs of

progress lists and continuations. A program starts with an empty event context.

$$\Delta ::= \epsilon \mid \Delta,\ v_e \mapsto (v_p,\ \lambda x.e)$$

In particular, $\Delta,\ \langle v \rangle$ may be reduced when $\Delta \neq \emptyset$, while $\Delta,\ v$ cannot.

The evaluation context $\mathcal{E}$, described in Figure 24, represents a family of terms containing a hole $[\cdot]$. If $\mathcal{E}$ is an evaluation context, then $\mathcal{E}[e]$ represents $\mathcal{E}$ with the term e substituted for the hole. The evaluation context and the rule (E-Congruence) specify the evaluation order of subexpressions.

$\mathcal{E}$	::=	$[\cdot]$	expression hole
	$\mid$	$\langle \mathcal{E} \rangle$	system expression
	$\mid$	$\mathcal{E}\ e \mid v\ \mathcal{E} \mid v_f\ \mathcal{E}$	application
	$\mid$	$\mathcal{E};\ e$	sequence
	$\mid$	$(v_i^{i \in 1..k-1},\ \mathcal{E},\ e_i^{i \in k+1..n})$	tuples
	$\mid$	$\mathcal{E}[j]$	projection
	$\mid$	$\ell(\mathcal{E})$	tagged expression
	$\mid$	$\texttt{adv}\ \mathcal{E}$	event expressions
	$\mid$	$\texttt{async}\ v_e\ v_p\ \mathcal{E}$	
	$\mid$	$\textbf{case}\ \mathcal{E}\ \textbf{of}\ \{\ \overline{\ell(x) \Rightarrow e}\ \}$	case expression
	$\mid$	$\mathcal{E} \bullet (e,\ v_p,\ e_k,\ e_h)$	arrow application
	$\mid$	$v_a \bullet (\mathcal{E},\ v_p,\ e_k,\ e_h)$	
	$\mid$	$v_a \bullet (v,\ v_p,\ \mathcal{E},\ e_h)$	
	$\mid$	$v_a \bullet (v,\ v_p,\ v_k,\ \mathcal{E})$	

Figure 24: Evaluation context.

Figures 25 and 26 define the operational semantics of abstract arrows. The rule (E-Seq) encodes sequencing which evaluates left-to-right. The rule (E-Case) shows that case expressions reduce to the arm labeled with the tag of the value being matched. Evaluation may become *stuck* if no such arm exists.

The rule (E-Arrow-App) is straightforward, as arrow application is simply a sugared form of lambda application.

E-Congruence

$$\frac{\Delta,\ e \to \Delta',\ e'}{\Delta,\ \mathcal{E}[e] \to \Delta',\ \mathcal{E}[e']}$$

E-App

$$\Delta,\ (\lambda x.e)\ v \to \Delta,\ [v/x]e$$

E-Seq

$$\Delta,\ v;\ e \to \Delta,\ e$$

E-Proj

$$\frac{1 \le j \le n}{\Delta,\ (v_i)^{i \in 1..n}[j] \to \Delta,\ v_j}$$

E-Case

$$\Delta,\ \textbf{case}\ \ell_i(v)\ \textbf{of}\ \{\ell_i(x_i) \Rightarrow e_i\}^{i \in 1..n} \to \Delta,\ [v/x_i]e_i$$

Figure 25: Operational semantics (lambda calculus and extensions).

We distinguish the execution of a host function from application of lambda terms (E-Host-App). The application of a host function may produce a value or raise an exception. The syntax in Section 5.1 requires that host application be read via case expression, ensuring that both normal and exceptional values are handled appropriately. We also insert a runtime type check to ensure the output of a call to f is 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. The rule (E-Fix) is a standard reduction rule for a fixed-point combinator.

Rules (E-Async) and (E-Event) describe the semantics of JavaScript's single-threaded event loop. JavaScript executes a chunk of code until completion before selecting a callback function to invoke from a queue determined by events which have been triggered externally. Rule (E-Async) adds a mapping from the event object v_e to a pair of a progress list a callback function to the event context Δ. Rule (E-Event) can be applied once there is no reducible term, and there is some event $v_e \in \Delta$ which has been triggered *externally* and *concurrently* (e.g. timer elapsed, user interface interaction, or network communication completed). The response of the event, either nominal or exceptional, is retrieved by the function `Resp`. The response is then applied to the continuation, and the continuation body becomes the current reducible term. This

E-Arrow-App

$$\frac{v_a = \lambda x.\lambda p.\lambda k.\lambda h.e_0 \qquad v_k = \lambda y.e_1 \qquad v_h = \lambda z.e_2}{\Delta,\ v_a \bullet (v,\ v_p,\ v_k,\ v_h) \to \Delta,\ [v/x,\ v_p/p,\ v_k/k,\ v_h/h]e_0}$$

E-Host-App

$$\frac{(v_f\ v) \downarrow v' \qquad \texttt{Annot}_\texttt{F}(v_f) = \tau_1 \to \tau_2 \setminus (C,\ E) \qquad \emptyset \vdash v' : \langle succ : \tau_2,\ fail : \tau_3 \rangle \qquad \tau_3 \in E \text{ or } (E = \emptyset \text{ and } \tau_3 = \top)}{\Delta,\ v_f\ v \to \Delta,\ v'}$$

E-Fix

$$\Delta,\ \texttt{fix}(\lambda\omega.e) \to \Delta,\ [\texttt{fix}(\lambda\omega.e)/\omega]e$$

E-Async

$$\Delta,\ \texttt{async}\ v_e\ v_p\ \lambda x.e \to \Delta \cup \{v_e \mapsto (v_p,\ \lambda x.e)\},\ ()$$

E-Event

$$\frac{v_e = \texttt{evn}(v,\ \tau_1 \setminus C,\ E) \qquad v_e \mapsto (v_p,\ \lambda x.e) \in \Delta \qquad \emptyset \vdash \texttt{Resp}(v_e) : \langle succ : \tau_1,\ fail : \tau_2 \rangle \qquad \tau_2 \in E \text{ or } (E = \emptyset \text{ and } \tau_2 = \top)}{\Delta,\ \langle v \rangle \to \Delta \setminus \{v_e \mapsto (v_p,\ \lambda x.e)\},\ \langle [\texttt{Resp}(v_e)/x]e \rangle}$$

E-Advance

$$\Delta,\ \texttt{adv}\ (P_i^j :: v_p) \to \{v_e \mapsto (v_p',\ \lambda x.e) \in \Delta \mid P_i^k \notin v_p',\ k \neq j\},\ \texttt{adv}\ v_p$$

E-Advance-Quiet

$$\Delta,\ \texttt{adv}\ (Q_i :: v_p) \to \Delta,\ ()$$

E-Advance-Empty

$$\Delta,\ \texttt{adv}\ \epsilon \to \Delta,\ ()$$

Figure 26: Operational semantics (host application and events).

term is evaluated to completion before the rule can be applied again. It is assumed that when rule (E-Event) is applied that v_e has completed - evaluation otherwise blocks.

The rules (E-Advance) and (E-Advance-Empty) describe a specific form of event cancellation used by abstract arrows. The expression $\texttt{adv}\ v_p$ will prune from the event context Δ all registered event handlers which are associated with a progress token that does **not** occur in v_p. Essentially, when the arrow associated with progress list v_p makes progress, other arrows waiting on an event

have *lost* the race and must be canceled. We cancel each progress token in the list recursively, as `any` combinators may be deeply nested, and a single progress event may cause multiple `any` combinators to choose a *winning arm* (arrow a_*).

The addition of rule (E-ADVANCE-QUIET) enables the `noemit` combinator, but changes the behavior of `adv` v_p subtly. The translation rules in Section 5.3 show that `noemit` combinators immediately introduce a Q_i progress token. When this progress token is encountered while advancing a progress list, the cancellation halts. This confines cancellations to the *subtree* rooted at the nearest Q_i, and only arrows associated with progress tokens within the same subtree are affected.

The arrows of Figure 23 help demonstrate this difference: if the progress list associated with arrow b is advanced, then arrows a, c, and d are all canceled; however, if the progress list associated with arrow d is advanced, then arrow c is canceled but arrows a and b are not.

5.3. Translation to Abstract Syntax

$$
\begin{aligned}
[\![\texttt{lift}(f)]\!] &\equiv \lambda x.\lambda p.\lambda k.\lambda h.\ \textbf{case}\ f\ x\ \textbf{of}\ \mathit{succ}(y) \Rightarrow k\ y,\ \mathit{succ}(y) \Rightarrow h\ y \\
[\![\texttt{ajax}(c)]\!] &\equiv \lambda x.\lambda p.\lambda k.\lambda h.\ \textbf{case}\ c\ x\ \textbf{of} \\
&\qquad \mathit{succ}(y) \Rightarrow \texttt{async ajaxEv}(y,\ \tau \setminus C,\ E)\ p\ \lambda v.\textbf{case}\ v\ \textbf{of} \\
&\qquad\qquad \mathit{succ}(z) \Rightarrow \texttt{adv}\ p;\ k\ z, \\
&\qquad\qquad \mathit{fail}(z) \Rightarrow h\ z, \\
&\qquad \mathit{fail}(y) \Rightarrow h\ y \\
&\qquad\quad \triangleright \text{ where } \texttt{Annot}_V(c) = \tau \setminus C \text{ and } E = \{\mathit{AjaxError}\} \\
[\![\texttt{delay}(n)]\!] &\equiv \lambda x.\lambda p.\lambda k.\lambda h.\ \texttt{async timeEv}(n,\ \top,\ \emptyset)\ p\ \lambda v.\textbf{case}\ v\ \textbf{of} \\
&\qquad \mathit{succ}(y) \Rightarrow \texttt{adv}\ p;\ k\ x,\ \mathit{fail}(y) \Rightarrow \texttt{()} \\
[\![a.\texttt{run}()]\!] &\equiv [\![a]\!] \bullet (\texttt{()},\ \epsilon,\ \lambda.\texttt{()},\ \lambda.\texttt{()})
\end{aligned}
$$

Figure 27: Arrow translation rules (constructors).

Figures 27 and 28 define the translation from concrete arrows to abstract arrows. 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 `domelem`, `split`, and `nth` arrows as they can be translated from simple lifted functions. To further reduce clutter, the semantics do not include the `domevent` constructor, as it is only trivially different from `delay` and `ajax` in relevant semantics.

The arrow `lift`(f) is translated to an expression that invokes the host function f, then applies the result of the function to the continuation k on success, and to h on exception. 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 async point.

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

The translation rules for the `seq`, `product`, and `try` combinators are fairly straightforward. The combinator `any`(a_1, a_2) is translated to two abstract arrows $[\![a_1]\!]$ and $[\![a_2]\!]$ with diverging progress lists. The two progress lists ensure that if the execution of a_1 makes progress, then a_2 is canceled (and the opposite). The combinator `noemit`(a) is translated to an abstract arrow $[\![a]\!]$ with a progress list prefixed by a sole progress token, which acts as a cancellation boundary for the executing arrow.

6. Properties

This section sketches a proof of soundness for typed arrows. The proof is based on a pair of progress and preservation theorems for arrows translated to abstract syntax [6]. Full proofs for each stated theorem appears in the appendix.

First, we establish that the translation of concrete arrows into abstract syntax preserves arrow types (Theorem 1, stated below). For abstract syntax, we

[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.

$$
\begin{aligned}
[\![\texttt{seq}(a_1, a_2)]\!] &\equiv \lambda x.\lambda p.\lambda k.\lambda h. \\
&\qquad [\![a_1]\!] \bullet (x,\ p,\ \lambda y.[\![a_2]\!] \bullet (y,\ p,\ k,\ h),\ h) \\
[\![\texttt{all}(a_1, a_2)]\!] &\equiv \lambda x.\lambda p.\lambda k.\lambda h. \\
&\qquad [\![a_1]\!] \bullet (x[1],\ p,\ \lambda y.[\![a_2]\!] \bullet (x[2],\ p,\ \lambda z.k\ (y, z),\ h),\ h) \\
[\![\texttt{try}(a, a_s, a_f)]\!] &\equiv \lambda x.\lambda p.\lambda k.\lambda h.\ [\![a]\!] \bullet (x,\ p, \\
&\qquad \lambda y.[\![a_s]\!] \bullet (y,\ p,\ k,\ h), \\
&\qquad \lambda y.[\![a_f]\!] \bullet (y,\ p,\ k,\ h)) \\
[\![\texttt{any}(a_1, a_2)]\!] &\equiv \lambda x.\lambda p.\lambda k.\lambda h. \\
&\qquad [\![a_1]\!] \bullet (x,\ P_i^1 :: p,\ k,\ h);\ [\![a_2]\!] \bullet (x,\ P_i^2 :: p,\ k,\ h) \\
[\![\texttt{noemit}(a)]\!] &\equiv \lambda x.\lambda p.\lambda k.\lambda h.\ [\![a]\!] \bullet (x,\ Q_i :: p,\ \lambda y.\texttt{adv}\ p;\ k\ y,\ h) \\
[\![\texttt{fix}(\omega\ \texttt{=>}\ a)]\!] &\equiv \texttt{fix}(\lambda\omega.[\![a]\!]) \\
[\![\omega]\!] &\equiv \omega
\end{aligned}
$$

Figure 28: Combinator translation rules (combinators). P_i^1, P_i^2, Q_i are fresh.

define an additional set of types composed of arrow value types and functions over arrow value types as follows.

$$\vec{\tau} ::= \tau \ \mid\ \vec{\tau} \to \vec{\tau}$$

This additional family of types is necessary to describe, in particular, the type of functions which accept continuations as an argument. A translated arrow has the following type

$$\tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top \setminus C$$

where τ_1 is the input to the arrow, τ_p is the type of all progress expressions e_p, $(\tau_2 \to \top)$ denotes the success continuation, and $(\tau_3 \to \top)$ denotes the failure continuation. The arrow, success continuation, and failure continuation do not return useful values as they may execute asynchronously.

Note that the type τ_3 can be a freshly generated type variable that serves as the type upper bound for the types of exceptions that may be thrown by an arrow. In the typing rules for arrows, this type is not required since the

dataflow of exceptions is implicit but the type must be provided explicitly after translation for the failure continuations.

We use $\hat{\Gamma}$ to denote the typing context for arrows translated to abstract syntax, defined from the typing context for concrete arrows Γ as follows.

$$\hat{\Gamma} = \{\omega : \alpha \to \tau_p \to (\beta \to \top) \to (\tau_3 \to \top) \to \top \mid (\omega : \alpha \rightsquigarrow \beta \in \Gamma)\}$$

The typing rules for arrows in abstract syntax are shown in Figures 29 and 30. The typing rules have the implicit assumption that all type variables are fresh and if $\hat{\Gamma} \vdash e : \overrightarrow{\tau} \setminus C$, then C is consistent.

We then show that the execution of an arrow translated to abstract syntax does not get stuck (Theorem 4, stated below). We make the assumption that all events occur eventually, therefore the event handlers added to Δ are invoked eventually.

Definition 1 (Well-formed event context)**.** Δ is well-formed if and only if for each $v_e \mapsto (v_p,\ \lambda x.e) \in \Delta$, $\emptyset \vdash v_e : \langle succ : \tau_1,\ fail : \tau_2\rangle \setminus C$, $\emptyset \vdash \lambda x.e : \tau_3 \to \top \setminus C'$, and $C \cup C' \cup \{\langle succ : \tau_1,\ fail : \tau_2\rangle \leq \tau_3\}$ is consistent.

Theorem 1 (Preservation of types under translation)**.** If a is well-typed with respect to a typing context Γ, then $[\![a]\!]$ has a symmetric type with respect to a typing context $\hat{\Gamma}$. Formally,

$$\begin{aligned} &\Gamma \vdash a : \tau_1 \rightsquigarrow \tau_2 \setminus (C,\ E) \\ \implies &\hat{\Gamma} \vdash [\![a]\!] : \tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top \setminus \hat{C} \end{aligned}$$

where $C_E = C \cup \{\tau \leq \tau_3 \mid \tau \in E\}$ and $closure(C_E) \subseteq closure(\hat{C})$.

Theorem 2 (Preservation of Δ)**.** If $\Delta,\ \hat{e} \to \Delta',\ \hat{e}',\ \Delta$ is well-formed, and $\Gamma \vdash \hat{e} : \overrightarrow{\tau} \setminus C$, then Δ' is well-formed.

Theorem 3 (Preservation of $\hat{\Gamma}$)**.** If $\Delta,\ \hat{e} \to \Delta',\ \hat{e}',\ \Delta$ is well-formed, and $\hat{\Gamma} \vdash \hat{e} : \overrightarrow{\tau} \setminus C$, then $\exists \overrightarrow{\tau}' \setminus C'$ such that $\hat{\Gamma} \vdash \hat{e}' : \overrightarrow{\tau}' \setminus C'$ and one of the following conditions hold.

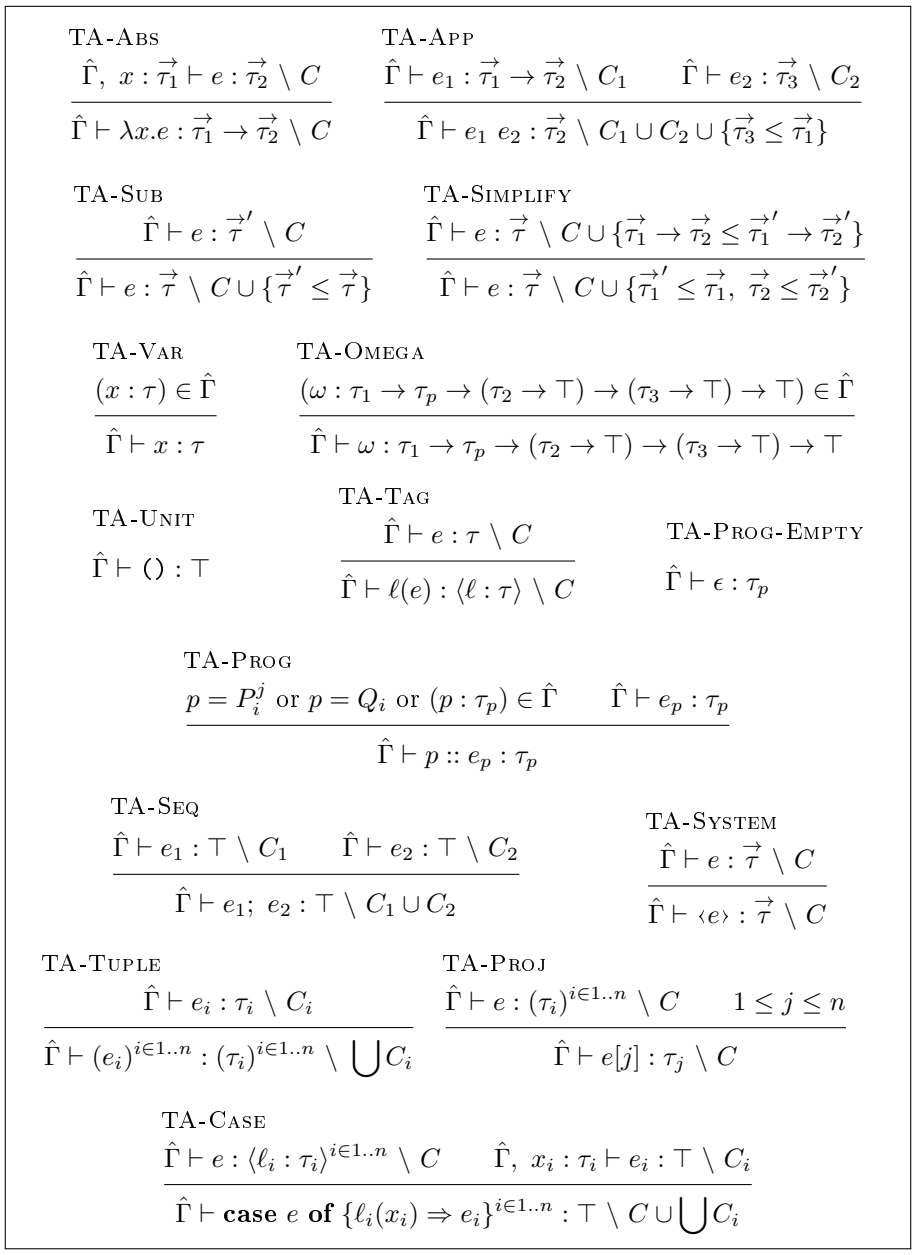


Figure 29: Typing rules for expressions in abstract syntax (lambda calculus and extensions).

1. $\hat{e} = \langle e \rangle$,

2. $\hat{e} \neq \langle e \rangle$, $\vec{\tau} = \vec{\tau}'$ and $closure(C') \subseteq closure(C)$, or

TA-Arrow

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

TA-Fix

$$\frac{\mathrm{T} \equiv \tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top \qquad \hat{\Gamma},\ \omega : \mathrm{T} \vdash e_a : \mathrm{T} \setminus C}{\hat{\Gamma} \vdash \texttt{fix}(\lambda\omega.e_a) : \mathrm{T} \setminus C}$$

TA-Arrow-App

$$\frac{\hat{\Gamma} \vdash e_a : \tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top \setminus C \qquad \hat{\Gamma} \vdash e : \tau_1 \setminus C_1 \qquad \hat{\Gamma} \vdash e_p : \tau_p \qquad \hat{\Gamma} \vdash e_k : \tau_2 \to \top \setminus C_2 \qquad \hat{\Gamma} \vdash e_h : \tau_3 \to \top \setminus C_3}{\hat{\Gamma} \vdash e_a \ \bullet\ (e,\ e_p,\ e_k,\ e_h) : \top \setminus C \cup C_1 \cup C_2 \cup C_3}$$

TA-Host-App

$$\frac{\texttt{Annot}_{\mathrm{F}}(v_f) = \tau_1 \to \tau_2 \setminus (C,\ E) \qquad \hat{\Gamma} \vdash e : \tau_1 \setminus C_1}{\hat{\Gamma} \vdash v_f\ e : \langle succ : \tau_2,\ fail : \alpha \rangle \setminus C \cup C_1 \cup \{\tau \leq \alpha \mid \tau \in E\}}$$

TA-Async

$$\frac{\hat{\Gamma} \vdash v_e : \langle succ : \tau_1,\ fail : \tau_2 \rangle \setminus C_1 \qquad \hat{\Gamma} \vdash e_p : \tau_p \qquad \hat{\Gamma} \vdash \lambda x.e : \tau_3 \to \top \setminus C_2}{\hat{\Gamma} \vdash \texttt{async}\ v_e\ e_p\ \lambda x.e\ :\ \top \setminus C_1 \cup C_2 \cup \{\langle succ : \tau_1,\ fail : \tau_2 \rangle \leq \tau_3\}}$$

TA-Ajax-Event

$$\frac{\hat{\Gamma} \vdash v : \{url : String\}}{\hat{\Gamma} \vdash \texttt{ajaxEv}(v,\ \tau \setminus C,\ \{AjaxError\}) : \langle succ : \tau,\ fail : AjaxError \rangle \setminus C}$$

TA-Time-Event

$$\frac{\hat{\Gamma} \vdash e : Number}{\hat{\Gamma} \vdash \texttt{timeEv}(e,\ \top,\ \emptyset) : \langle succ : \top,\ fail : \top \rangle}$$

TA-Advance

$$\frac{\hat{\Gamma} \vdash e_p : \tau_p}{\hat{\Gamma} \vdash \texttt{adv}\ e_p : \top}$$

Figure 30: Typing rules for expressions in abstract syntax (arrow definitions, host application, and events).

3. $\hat{e} \neq \text{‹}e\text{›}$, $\vec{\tau} \neq \vec{\tau}'$ and $closure(C' \cup \{\vec{\tau}' \leq \vec{\tau}\}) \subseteq closure(C)$.

Theorem 4 (Progress)**.** If $\emptyset \vdash \hat{e} : \overrightarrow{\tau} \setminus C$ and Δ is well-formed, then either one of the following conditions hold.

1. $\hat{e} = v$,
2. $\hat{e} = \langle v \rangle$ and $\Delta = \emptyset$,
3. $\exists \Delta',\ \hat{e}'$ such that $\Delta,\ \hat{e} \to \Delta',\ \hat{e}'$, or
4. a typing premise fails in rule (E-Host-App) or (E-Event).

7. 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. The full source is available at the project homepage[3].

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 the composite arrow

$$\texttt{selectOne} :: \mathit{DomElem} \rightsquigarrow \top$$

which blocks until a user selects a face-down card, then animates the card being revealed. It is reasonable to assume that running `selectOne` twice in a row will give us the desired functionality of selecting exactly two distinct cards. However, sequencing the arrow with itself results in a rightful type-clash, as the second invocation of `selectOne` expects a value of type *DomElem*, but the result of the first invocation is the insignificant value *undefined* (with type $\top$).

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

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

was encoded as a *derived combinator* in our library.

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

Such idioms were not uncommon, and each of them could be easily expressed by the composition of *core* constructors and combinators defined in Section 3.1 and Section 3.2.

In the following, we refer to a set of sample applications. This set includes the example in Section 2 and the game *Memory*, described above. This set also includes a more complex version of the paging example which adds pagination and pre-fetching of results from a remote server, and an example which animates the execution of Knuth Shuffle and Bubble Sort algorithms.

Annotation Burden. We counted the number of arrow annotations in a set of sample applications described above. The results are shown in Figure 7.

	Arrows	**Annotations**	**User Annotations**
Autocomplete	31	10	5
Paging	80	20	8
Memory	133	35	8
Animation	148	42	4

The count of arrows are the *total* number of arrows created during composition. The count of annotations is the number of arrows created from lifted functions or ajax configuration functions. The count of user annotations exclude the arrows annotated in the 'standard library'. The annotation burden required by developers seems to be minimal and only requires adding a single comment line to functions lifted to arrows.

Inference Overhead. We measured the overhead of arrow type inference a set of sample applications described above. The results are shown in Figure 7.

The measurements shown are averaged over 2,000 runs in Chrome (V8) with 1,000 warm-up runs discarded (which populated the annotation cache). While there is an increase in runtime, the cost is paid only once at application startup.

	Arrows	Disabled (ms)	Enabled (ms)
Autocomplete	31	0.312	2.988
Paging	80	0.756	7.118
Memory	133	1.388	12.951
Animation	148	1.577	10.317

Runtime Overhead. We measured the runtime overhead of runtime type-checking at the boundaries of lifted functions in a micro-benchmark. This benchmark created an arrow which rotates the values of a record with the following variable-sized type.

$$\{\ell_0 : \alpha,\ \ell_1 : \beta,\ \ell_2 : \gamma,\ \ldots,\ \ell_n : \omega\} \rightsquigarrow \{\ell_0 : \beta,\ \ell_1 : \gamma,\ \ldots, \ell_n : \alpha\}$$

This arrow was composed with itself a variable number of times with a `delay`(0) arrow composed in between to avoid stack limits during execution. The initial value of this arrow is given the following type, where each array contains 100 elements.

$$\{\ell_1 : [\text{“}v1\text{”}, \text{“}v2\text{”}, \ldots],\ \ell_2 : [1, 2, \ldots],\ \ell_3 : [\text{“}v1\text{”}, \text{“}v2\text{”}, \ldots],\ \ell_4 : [1, 2, \ldots],\ \ldots\}$$

The results of this benchmark are shown in Figure 7. Each cell contains the complete time the arrow takes to run with runtime type checking disabled (first) and enabled (second). These timings do not take into account composition overhead.

		# of Fields in Record			
		8	**16**	**32**	**64**
# of Arrows	**32**	0.197/0.255ms	0.131/0.374ms	0.186/0.685ms	0.342/1.575ms
	64	0.212/0.274ms	0.212/0.473ms	0.267/0.750ms	1.036/1.747ms
	128	0.370/0.624ms	0.377/0.654ms	0.422/0.968ms	2.092/1.705ms
	256	0.926/0.933ms	0.665/0.828ms	0.708/1.448ms	1.472/1.908ms

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.

8. Related Work

Arrows. Arrows [2] were first formalized as a generalization of monads [3]. As monadic type $m\ a$ represents a computation delivering an a, the arrow type $a\ b\ c$ represents a computation with input of type b delivering a c. Arrows are formally defined by extending simply-typed lambda calculus with three constants satisfying nine laws. The constants arr and $(\ggg)$ are lift and compose operations, respectively. The constant $first$ converts an arrow from b to c into an arrow on pairs. Our `lift` constructor and `seq` and `all` combinators encompass all three operations.

Alternate arrow calculi exist with only four constants satisfying five obvious laws [7, 8]. The laws of these calculus follow from beta and eta laws of the lambda calculus and fit into well-known patterns. The calculi are equivalent, and a translation scheme from one calculus to the other exists.

Arrowlets. Arrowlets is a JavaScript library for using arrows [1], 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.

Functional Reactive Programming (FRP). FRP [9] has seen an emergence in JavaScript. FRP is a *data-flow oriented* paradigm in which changes to values are propagated through the system, allowing it to react to external events.

Elm [10] and Flapjax [11] are two reactive languages that use JavaScript as a compilation target. Elm adds *asynchronicity* to FRP, allowing the developer to specify potentially long-running signal computation so they may be computed asynchronously. This prevents time leaks in the system which would make the GUI appear unresponsive. Flapjax is structured around the *event stream* abstraction, which allows for inter-program communication as well as communication with external services. This makes Flapjax *data-flow oriented* in its composition, whereas arrows are defined to be *control-flow oriented*.

The use of both FRP and arrows have also been explored. Programming with FRP signal can often lead to conspicuous time and space leaks. If signals are defined in such a way that they model arrows (and obey the arrow laws), these leaks can be avoided completely [12].

Event Programming. The JavaScript ecosystem has seen the need for improving event programming idioms and has responded accordingly. jQuery 1.5 introduced *deferred objects* and other popular JavaScript framework and libraries have popularized *promises*. These objects represent values which may not yet be *resolved*. Callbacks can be composed and registered to be invoked when the desired value is actually produced. While it is a successful pattern for asynchronous callbacks, it is less powerful than the composition arrows which can create arbitrary control flow graphs through combinators. However, promises allow callbacks to be registered to multiple *states* (e.g. success, failure, done) of the deferred object or promise, so that different control flow paths can be taken depending on the outcome of the asynchronous operation.

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 encodes 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.

Coincidentally, 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 [13] 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.

Semantics. Our semantics were intended to cover *only* the interesting aspects of JavaScript's event loop in relation to arrow composition and execution. Maffeis et al. present a small-step operational semantics strictly conforming to the ECMAScript (ECMA262-3) specification [14]. These semantics cover *most* of the semantically interesting parts of JavaScript (leaving out constructs which are not insightful, e.g. `switch` and `for`, and regular expression matching), including heap objects and prototype lookup. Similarly, a core calculus for JavaScript, dubbed λ_{JS} was developed alongside a small-step operational semantics and a *desugaring* algorithm which translates JavaScript into the core calculus, showing equivalence [15].

9. Conclusion

We have presented an arrows library which encodes semantics 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.

References

[1] Y. P. Khoo, M. Hicks, J. S. Foster, V. Sazawal, Directing JavaScript with arrows, in: Proceedings of the 5th Symposium on Dynamic Languages, DLS '09, ACM, New York, NY, USA, 2009, pp. 49–58.

[2] J. Hughes, Generalising monads to arrows, Science of Computer Programming 37 (1998) 67–111.

[3] P. Wadler, The essence of functional programming, in: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1992, pp. 1–14.

[4] E. Fritz, T. Zhao, Type inference of asynchronous arrows in JavaScript, Reactive and Event-based Languages & Systems (2015).

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

[6] A. K. Wright, M. Felleisen, A syntactic approach to type soundness, Information and computation 115 (1) (1994) 38–94.

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

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

[9] Z. Wan, P. Hudak, Functional reactive programming from first principles, in: ACM SIGPLAN Notices, Vol. 35, ACM, 2000, pp. 242–252.

[10] E. Czaplicki, S. Chong, Asynchronous functional reactive programming for guis, in: ACM SIGPLAN Notices, Vol. 48, ACM, 2013, pp. 411–422.

[11] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi, Flapjax: a programming language for ajax applications, in: ACM SIGPLAN Notices, Vol. 44, ACM, 2009, pp. 1–20.

[12] P. Hudak, A. Courtney, H. Nilsson, J. Peterson, Arrows, robots, and functional reactive programming, in: Advanced Functional Programming, Springer, 2003, pp. 159–187.

[13] S. K. Muller, W. A. Duff, U. A. Acar, Practical abstractions for concurrent interactive programming, Tech. rep., Carnegie Mellon University (2015).

[14] S. Maffeis, J. C. Mitchell, A. Taly, An operational semantics for JavaScript, in: Programming languages and systems, Springer, 2008, pp. 307–325.

[15] A. Guha, C. Saftoiu, S. Krishnamurthi, The essence of JavaScript, in: ECOOP 2010–Object-Oriented Programming, Springer, 2010, pp. 126–150.

Appendix A. Type Simplification

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

Definition 2 (Bound Minimal)**.** An arrow type $\tau_{in} \rightsquigarrow \tau_{out} \setminus (C,\ E)$ 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.

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

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

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

$(\vee\ \mathrm{T})$ 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 *Number* and a lower bound of *String*.

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

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

Definition 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 $\alpha \in \tau$ and a type variable $\beta \in \tau'$, 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 E, or if α lower-bounds some type variable β^+ or $\beta^\pm$.

Definition 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 type 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 conditions hold.

1. $\{\tau \leq \alpha,\ \alpha \leq \tau\} \subseteq C$,
2. $\alpha^- \leq \tau \in C$ and $\alpha \leq \tau' \notin C\ \forall \tau' \neq \tau$, or
3. $\tau \leq \alpha^+ \in C$ and $\tau' \leq \alpha \notin C\ \forall \tau' \neq \tau$.

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 that is being substituted.

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.

We substitute 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.

Uls-Top	Uls-Self	Uls-Upper	Uls-Lower
$\tau \leq \top$	$\tau \leq \tau$	$\alpha^+ \leq \tau$	$\tau \leq \alpha^-$

Uls-NonVar	Uls-LowerUnknown	Uls-UpperUnknown
$\dfrac{\tau \notin A \qquad \tau' \notin A}{\tau \leq \tau'}$	$\dfrac{\alpha \notin \tau_{in} \qquad \alpha \notin \tau_{out}}{\alpha \leq \beta}$	$\dfrac{\alpha \notin \tau_{in} \qquad \alpha \notin \tau_{out}}{\beta \leq \alpha}$

Figure A.31: Useless constraint elimination rules.

Definition 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 A.31 are considered useless. A constraint set can be *pruned* by repeatedly removing all useless constraints.

Appendix B. Proof of Theorem 1 (See page 38)

Theorem 1 (Preservation of types under translation)**.** If a is well-typed with respect to a typing context Γ, then $[\![a]\!]$ has a symmetric type with respect to a typing context $\hat{\Gamma}$. Formally,

$$\begin{aligned} & \Gamma \vdash a : \tau_1 \rightsquigarrow \tau_2 \setminus (C,\ E) \\ \implies & \hat{\Gamma} \vdash [\![a]\!] : \tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top \setminus \hat{C} \end{aligned}$$

where $C_E = C \cup \{\tau \leq \tau_3 \mid \tau \in E\}$ and $closure(C_E) \subseteq closure(\hat{C})$.

Proof. We prove by case analysis on a.

Case $[a = \texttt{fix}(\omega \Rightarrow a')]$. Let T and T′ denote the following function types.

$$\begin{aligned} \mathrm{T} &= \tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top \\ \mathrm{T}' &= \tau_1' \to \tau_p \to (\tau_2' \to \top) \to (\tau_3' \to \top) \to \top \end{aligned}$$

By rule (T-Fix), we have $\Gamma,\ \omega : \tau_1 \rightsquigarrow \tau_2 \vdash \alpha' : \tau_1' \rightsquigarrow \tau_2' \setminus (C',\ E)$ where $C = C' \cup \{\tau_1 \leq \tau_1',\ \tau_2' \leq \tau_2\}$ and by the induction hypothesis on a', we have $\hat{\Gamma},\ \omega : \mathrm{T} \vdash [\![a']\!] :\ \mathrm{T}' \setminus \hat{C}'$ where $C' \cup \{\tau \leq \tau_3' \mid \tau \in E\} \subseteq \hat{C}'$. We can type the translation by rule (TA-Fix) as follows. The rules (TA-Sub) and (TA-Simplify) are used to unify the type of $[\![a']\!]$ and T.

$$\hat{\Gamma} \vdash \texttt{fix}(\lambda\omega.[\![a']\!]) : \mathrm{T} \setminus \hat{C}' \cup \underbrace{\{\tau_1 \leq \tau_1',\ \tau_2' \leq \tau_2,\ \tau_3' \leq \tau_3\}}_{\text{simplified from } \mathrm{T}' \leq \mathrm{T}}$$

Thus, $closure(C_E \cup \{\tau \leq \tau_3' \mid \tau \in E\} \cup \{\tau_3' \leq \tau_3\}) \subseteq closure(\hat{C})$.

Case $[a = \omega]$. By rule (T-Omega), $\hat{\Gamma} \vdash \omega : \tau_1 \rightsquigarrow \tau_2$ where $C = E = \emptyset$. By rule (TA-Omega) and definition of $\hat{\Gamma}$, we have the following for some τ_3.

$$\hat{\Gamma} \vdash \omega : \tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top$$

For the remaining cases, $[\![a]\!] \equiv \lambda x.\lambda p.\lambda k.\lambda h.e$ for some x, p, k, h, and e. Thus, it is sufficient to show the following.

$$\hat{\Gamma},\ (x : \tau_1),\ (p : \tau_p),\ (k : \tau_2 \to \top),\ (h : \tau_3 \to \top) \vdash e : \top \setminus \hat{C}$$

For convenience, we define Γ' to be the typing context which which the arrow body is typed, i.e. let $\Gamma' = \hat{\Gamma},\ (x : \tau_1),\ (p : \tau_p),\ (k : \tau_2 \to \top),\ (h : \tau_3 \to \top)$.

Case $[a = \texttt{lift}(f)]$. By rule (T-Lift), $\texttt{Annot}_\texttt{F}(f) = \tau_1 \to \tau_2 \setminus (C,\ E)$. The remainder of this case follows directly by deriving a type for the translated arrow body e by rules (TA-Host-App), (TA-Case), and (TA-App).

$$\Gamma' \vdash \textbf{case}\ (f\ x)\ \textbf{of}\ \mathit{succ}(y) \Rightarrow k\ y,\ \mathit{fail}(y) \Rightarrow h\ y : \top \setminus C \cup \underbrace{\{\tau \leq \alpha \mid \tau \in E\}}_{\text{from typing of } (f\ x)} \cup \{\alpha \leq \tau_3\}$$

Case $[a = \texttt{ajax}(c)]$. By rule (T-Ajax), $\texttt{Annot}_\texttt{F}(c) = \tau_1 \to \tau_U \setminus (C_1,\ E_1)$ and $\texttt{Annot}_\texttt{V}(c) = \tau_2 \setminus C_2$ where τ_U denotes the type $\{\mathit{url} : \mathit{String}\}$, $C = C_1 \cup C_2$, and $E = E_1 \cup \{\mathit{AjaxError}\}$. By rule (T-Host-App), $c\ x$ yields a value of type $\langle \mathit{succ} : \tau_U,\ \mathit{fail} : \alpha \rangle \setminus \{\tau \leq \alpha \mid \tau \in E_1\}$ with respect to Γ'. Now, we derive a type for the async callback body e' by rules (TA-Case), (TA-Advance), and (TA-App). This derivation assumes a type for the unbound variable v, which we unify in the following.

$$\Gamma',\ v : \langle \mathit{succ} : \tau_2,\ \mathit{fail} : \tau_3 \rangle \vdash e' = \textbf{case}\ v\ \textbf{of}\ \mathit{succ}(z) \Rightarrow \texttt{adv}\ p;\ k\ z,\ \mathit{fail}(z) \Rightarrow h\ z : \top$$

The remainder of this case follows directly by deriving a type for the translated arrow body e by rules (TA-Case), (TA-Async), (TA-Ajax-Event), and (TA-Abs).

$$\Gamma' \vdash \textbf{case}\ c\ x\ \textbf{of}\ \mathit{succ}(y) \Rightarrow \texttt{async}\ \overbrace{\texttt{ajaxEv}(y,\ \tau_2 \setminus C_2,\ \{\mathit{AjaxError}\})}^{\text{yields } \langle \mathit{succ} : \tau_2,\ \mathit{fail} : \mathit{AjaxError} \rangle \setminus C_2}\ p\ \lambda v.e',\ \mathit{fail}(y) \Rightarrow h\ y : \top \setminus \hat{C}$$

The constraint set above is equivalent to the following, where the additional constraints originate from application rules (TA-Sub) and (TA-Simplify) which are used to make the type of v consistent with its use as the async callback, and the type of h consistent with its argument y.

$$C_1 \cup C_2 \cup \{\overbrace{\mathit{AjaxError} \leq \tau_3}^{\text{from subsumption of } v},\ \underbrace{\alpha \leq \tau_3}_{\text{from simplifying } h}\} \cup \{\tau \leq \alpha \mid \tau \in E_1\}$$

Case $[a = \texttt{delay}(n)]$. By rule (T-Delay), n is a number and $C = E = \emptyset$. The remainder of this case follows directly by deriving a type for the translated arrow body e by rules (TA-Async), (TA-Time-Event), (TA-Abs), (TA-Case), (TA-Advance), (TA-App), and (TA-Unit).

$$\Gamma' \vdash \texttt{async}\ \underbrace{\texttt{timeEv}(n,\ \top,\ \emptyset)}_{\text{yields } \langle \mathit{succ} : \top,\ \mathit{fail} : \top \rangle}\ p\ \lambda v.\mathbf{case}\ v\ \mathbf{of}\ \mathit{succ}(y) \Rightarrow \texttt{adv}\ p;\ k\ x,\ \mathit{fail}(y) \Rightarrow \texttt{()} : \top \setminus \emptyset$$

Case $[a = \texttt{seq}(a_1, a_2)]$. By rule (T-Seq), the types of a_1 and a_2 are respectively $\tau_1 \rightsquigarrow \tau_1' \setminus (C_1,\ E_1)$ and $\tau_2' \rightsquigarrow \tau_2 \setminus (C_2,\ E_2)$ where $C = C_1 \cup C_2 \cup \{\tau_1' \leq \tau_2'\}$ and $E = E_1 \cup E_2$. By the induction hypothesis on a_1 and a_2,

$$\hat{\Gamma} \vdash [\![a_1]\!] : \tau_1 \to \tau_p \to (\tau_1' \to \top) \to (\tau_3^1 \to \top) \to \top \setminus \hat{C}_1$$
$$\hat{\Gamma} \vdash [\![a_2]\!] : \tau_2' \to \tau_p \to (\tau_2 \to \top) \to (\tau_3^2 \to \top) \to \top \setminus \hat{C}_2$$

where $C_1 \cup \{\tau \leq \tau_3^1 \mid \tau \in E_1\} \subseteq \hat{C}_1$ and $C_2 \cup \{\tau \leq \tau_3^2 \mid \tau \in E_2\} \subseteq \hat{C}_2$. The remainder of this case follows directly by deriving a type for the translated arrow body e by rules (TA-Arrow-App) and (TA-Abs).

$$\Gamma' \vdash [\![a_1]\!] \bullet (x,\ p,\ \lambda y.\ [\![a_2]\!] \bullet (y,\ p,\ k,\ h),\ h) : \top \setminus \hat{C}_1 \cup \hat{C}_2 \cup \{\overbrace{\tau_1' \leq \tau_2'}^{\text{from simplifying } \lambda y \text{ term}},\ \underbrace{\tau_3^1 \leq \tau_3,\ \tau_3^2 \leq \tau_3}_{\text{from simplifying } h}\}$$

The additional constraints originate from application of rules (TA-Sub) and (TA-Simplify) which are used to make the type of the abstraction consistent with its use as $[\![a_1]\!]$'s success callback, and the type of h consistent with its use in as failure callbacks of both $[\![a_1]\!]$ and $[\![a_2]\!]$. The constraint set above is a

superset of the following.

$$C_1 \cup C_2 \cup \{\tau \le \tau_3^1 \mid \tau \in E_1\} \cup \{\tau \le \tau_3^2 \mid \tau \in E_2\} \cup \{\tau_1' \le \tau_2',\ \tau_3^1 \le \tau_3,\ \tau_3^2 \le \tau_3\}$$

The missing constraints $\{\tau \le \tau_3 \mid \tau \in E\}$ are introduced via transitive closure rules.

Case $[a = \texttt{all}(a_1, a_2)]$. By rule (T-All), the types of a_1 and a_2 are respectively $\tau_1^1 \rightsquigarrow \tau_2^1 \setminus (C_1,\ E_1)$ and $\tau_1^2 \rightsquigarrow \tau_2^2 \setminus (C_2,\ E_2)$ where $C = C_1 \cup C_2$, $E = E_1 \cup E_2$, $\tau_1 = (\tau_1^1,\ \tau_1^2)$, and $\tau_2 = (\tau_2^1,\ \tau_2^2)$. By the induction hypothesis on a_1 and a_2,

$$\hat{\Gamma} \vdash [\![a_1]\!] : \tau_1^1 \to \tau_p \to (\tau_2^1 \to \top) \to (\tau_3^1 \to \top) \to \top \setminus \hat{C}_1$$
$$\hat{\Gamma} \vdash [\![a_2]\!] : \tau_1^2 \to \tau_p \to (\tau_2^2 \to \top) \to (\tau_3^2 \to \top) \to \top \setminus \hat{C}_2$$

where $C_1 \cup \{\tau \le \tau_3^1 \mid \tau \in E_1\} \subseteq \hat{C}_1$ and $C_2 \cup \{\tau \le \tau_3^2 \mid \tau \in E_2\} \subseteq \hat{C}_2$. The remainder of this case follows directly by deriving a type for the translated arrow body e by rules (TA-Arrow-App), (TA-Proj), (TA-Abs), (TA-App), and (TA-Tuple).

$$\Gamma' \vdash [\![a_1]\!] \bullet (x[0],\ p,\ \lambda y. \quad [\![a_2]\!] \bullet (x[1],\ p,\ \lambda z.k\ (y, z),\ h),\ h) : \top \setminus \hat{C}_1 \cup \hat{C}_2 \cup \overbrace{\{\tau_3^1 \le \tau_3,\ \tau_3^2 \le \tau_3\}}^{\text{from simplifying } h}$$

The additional constraints originate from application of rules (TA-Sub) and (TA-Simplify) which are used to make the type of h consistent with its use as the failure callback of both $[\![a_1]\!]$ and $[\![a_2]\!]$.

Case $[a = \texttt{try}(a_1, a_2, a_3)]$. By rule (T-Try), the types of a_1, a_2, and a_3 are respectively $\tau_1 \rightsquigarrow \tau_2^1 \setminus (C_1,\ E_1)$, $\tau_1^2 \rightsquigarrow \tau_2^2 \setminus (C_2,\ E_2)$, and $\tau_1^3 \rightsquigarrow \tau_2^3 \setminus (C_3,\ E_3)$ where $C = C_1 \cup C_2 \cup C_3 \cup \{\tau_2^1 \le \tau_1^2,\ \tau_2^2 \le \tau_2\ \tau_2^3 \le \tau_2\} \cup \{\tau \le \tau_1^3 \mid \tau \in E_1\}$ and $E = E_2 \cup E_3$. By the induction hypothesis on a_1, a_2, and a_3,

$$\hat{\Gamma} \vdash [\![a_1]\!] : \tau_1 \to \tau_p \to (\tau_2^1 \to \top) \to (\tau_3^1 \to \top) \to \top \setminus \hat{C}_1$$
$$\hat{\Gamma} \vdash [\![a_2]\!] : \tau_1^2 \to \tau_p \to (\tau_2^2 \to \top) \to (\tau_3^2 \to \top) \to \top \setminus \hat{C}_2$$
$$\hat{\Gamma} \vdash [\![a_3]\!] : \tau_1^3 \to \tau_p \to (\tau_2^3 \to \top) \to (\tau_3^3 \to \top) \to \top \setminus \hat{C}_3$$

where $C_1 \cup \{\tau \leq \tau_3^1 \mid \tau \in E_1\} \subseteq \hat{C}_1$, $C_2 \cup \{\tau \leq \tau_3^2 \mid \tau \in E_2\} \subseteq \hat{C}_2$, and $C_3 \cup \{\tau \leq \tau_3^3 \mid \tau \in E_3\} \subseteq \hat{C}_3$. The remainder of this case follows directly by deriving a type for the translated arrow body e by rules (TA-Arrow-App) and (TA-Abs).

$$\Gamma' \vdash [\![a_1]\!] \bullet (x,\ p,\ \lambda y.[\![a_2]\!] \bullet (y,\ p,\ k,\ h),\ \lambda y.[\![a_3]\!] \bullet (y,\ p,\ k,\ h)) : \top \setminus \hat{C}$$

The constraint set $\hat{C}$ is a superset of the following, where the additional constraints originate from application of rules (TA-Sub) and (TA-Simplify) which are used to make the type of the abstractions consistent with the their use as the success and failure callbacks of $[\![a_1]\!]$, and the type of k and h consistent with their use as success and failure callbacks, respectively, of $[\![a_2]\!]$ and $[\![a_3]\!]$.

$$C_1 \cup C_2 \cup C_3 \cup \{\tau \leq \tau_3^1 \mid \tau \in E_1\} \cup \{\tau \leq \tau_3^2 \mid \tau \in E_2\} \cup \{\tau \leq \tau_3^3 \mid \tau \in E_3\} \cup$$
$$\{\underbrace{\tau_2^2 \leq \tau_2,\ \tau_2^3 \leq \tau_2}_{\text{from simplifying } k}\} \cup \{\underbrace{\tau_3^2 \leq \tau_3,\ \tau_3^3 \leq \tau_3}_{\text{from simplifying } h}\} \cup \{\underbrace{\tau_2^1 \leq \tau_1^2,\ \tau_3^1 \leq \tau_1^3}_{\text{from simplifying } \lambda y \text{ terms}}\}$$

The missing constraints $\{\tau \leq \tau_3 \mid \tau \in E_2 \cup E_3\}$ are introduced via transitive closure rules.

Case $[a = \mathsf{any}(a_1, a_2)]$. By rule (T-All), the types of a_1 and a_2 are respectively $\tau_1^1 \rightsquigarrow \tau_2^1 \setminus (C_1,\ E_1)$ and $\tau_1^2 \rightsquigarrow \tau_2^2 \setminus (C_2,\ E_2)$ where $C = C_1 \cup C_2 \cup \{\tau_1 \leq \tau_1^1,\ \tau_1 \leq \tau_1^2,\ \tau_2^1 \leq \tau_2,\ \tau_2^1 \leq \tau_2\}$ and $E = E_1 \cup E_2$. By the induction hypothesis on a_1 and a_2,

$$\hat{\Gamma} \vdash [\![a_1]\!] : \tau_1^1 \to \tau_p \to (\tau_2^1 \to \top) \to (\tau_3^1 \to \top) \to \top \setminus \hat{C}_1$$
$$\hat{\Gamma} \vdash [\![a_2]\!] : \tau_2^2 \to \tau_p \to (\tau_2^2 \to \top) \to (\tau_3^2 \to \top) \to \top \setminus \hat{C}_2$$

where $C_1 \cup \{\tau \leq \tau_3^1 \mid \tau \in E_1\} \subseteq \hat{C}_1$ and $C_2 \cup \{\tau \leq \tau_3^2 \mid \tau \in E_2\} \subseteq \hat{C}_2$. The remainder of this case follows directly by deriving a type for the translated arrow body e by rules (TA-Seq), (TA-Arrow-App), and (TA-Prog).

$$\Gamma' \vdash [\![a_1]\!] \bullet (x,\ P_i^1 :: p,\ k,\ h);\ [\![a_2]\!] \bullet (x,\ P_i^2 :: p,\ k,\ h) : \top \setminus \hat{C}$$

The constraint set $\hat{C}$ is a superset of the following, where the additional constraints originate from application of rules (TA-Sub) and (TA-Simplify) which

are used to make the type of k and h consistent with its use as the success and failure callbacks of both $[\![a_1]\!]$ and $[\![a_2]\!]$, and the type of x consistent with its use as a parameter to both $[\![a_1]\!]$ and $[\![a_2]\!]$.

$$C_1 \cup \{\tau \le \tau_3^1 \mid \tau \in E_1\} \cup C_2 \cup \{\tau \le \tau_3^2 \mid \tau \in E_2\} \cup$$
$$\{\underbrace{\tau_1 \le \tau_1^1,\ \tau_1 \le \tau_1^2}_{\text{from subsumption of } x},\ \underbrace{\tau_2^1 \le \tau_2,\ \tau_2^2 \le \tau_2}_{\text{from simplifying } k},\ \underbrace{\tau_3^1 \le \tau_3,\ \tau_3^2 \le \tau_3}_{\text{from simplifying } h}\}$$

The missing constraints $\{\tau \le \tau_3 \mid \tau \in E_2 \cup E_3\}$ are introduced via transitive closure rules.

Case $[a = \texttt{noemit}(a')]$. By rule (T-NoEmit), $\Gamma' \vdash a' : \tau_1 \rightsquigarrow \tau_2 \setminus (C,\ E)$. By the induction hypothesis,

$$\hat{\Gamma} \vdash [\![a']\!] : \tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top \setminus \hat{C}'$$

where $C_E \subseteq \hat{C}'$. The remainder of this case follows directly by deriving a type for the translated body e by rules (TA-Arrow-App), (TA-Prog), (TA-Abs), (TA-Seq), (TA-Advance), and (TA-App).

$$\Gamma' \vdash [\![a']\!] \bullet (x,\ Q_i :: p,\ \lambda y.\texttt{adv}\ p;\ k\ y,\ h) : \top \setminus \hat{C}'$$

This completes case analysis on a, and the proof. □

Appendix C. Proof of Theorem 2 (See page 38)

Theorem 2 (Preservation of Δ)**.** If $\Delta,\ \hat{e} \to \Delta',\ \hat{e}',\ \Delta$ is well-formed, and $\Gamma \vdash \hat{e} : \overrightarrow{\tau} \setminus C$, then Δ' is well-formed.

Proof. We prove by case analysis on $\hat{e}$.

Case $[\mathcal{E}[e_0]]$. Δ' is well-formed by the induction hypothesis on e_0.

Case $[\texttt{async}\ v_e\ v_p\ \lambda x.e_0]$. By rule (TA-Async), we have the following.

$$\frac{\hat{\Gamma} \vdash v_e : \langle \mathit{succ} : \tau_1,\ \mathit{fail} : \tau_2 \rangle \setminus C_1 \qquad \hat{\Gamma} \vdash v_p : \tau_p \qquad \hat{\Gamma} \vdash \lambda x.e_0 : \tau_3 \to \top \setminus C_2}{\hat{\Gamma} \vdash \texttt{async}\ v_e\ v_p\ \lambda x.e_0 : \top \setminus C_1 \cup C_2 \cup \{\langle \mathit{succ} : \tau_1,\ \mathit{fail} : \tau_2 \rangle \leq \tau_3\}}$$

Each $v_e \in \Delta$ is well-formed and, by Definition 1, $v_e \mapsto (v_p,\ \lambda x.e_0)$ is well-typed. By rule (E-Async), $\Delta' = \Delta \cup \{v_e \mapsto (v_p,\ \lambda x.e_0)\}$. Therefore, Δ' is well-formed.

Cases $[\langle v \rangle]$ *and* $[\texttt{adv}\ (P_i^j :: v_p)]$. $\Delta' \subseteq \Delta$ by rules (E-Event) and (E-Async) for each case, respectively. Therefore, each $v_e \in \Delta'$ is well-typed and Δ' is well-formed.

In all other cases, $\Delta' = \Delta$ and the proof is trivial. This completes case analysis on e, and the proof. □

Appendix D. Proof of Theorem 3 (See page 38)

Lemma 1 (Substitution)**.** If $\hat{\Gamma},\ x : \overrightarrow{\tau}' \vdash \hat{e} : \overrightarrow{\tau} \setminus C$, $\hat{\Gamma} \vdash v : \overrightarrow{\tau}' \setminus C'$, and $C \cup C'$ is consistent, then $\hat{\Gamma} \vdash [v/x]\hat{e} : \overrightarrow{\tau} \setminus \hat{C}$ such that $C \subseteq \hat{C} \subseteq C \cup C'$.

Proof. For simplicity, we assume that each variable introduced (via abstraction, fix, and case) is done so only once. We prove by case analysis on $\hat{e}$.

Case $[x]$. The type of x is given by the typing context $\hat{\Gamma}$. As $[v/x]x = v$, the type of $[v/x]x$ is assumed, as follows.

$$\hat{\Gamma},\ x : \overrightarrow{\tau}' \vdash x : \overrightarrow{\tau}' \setminus \emptyset \qquad\qquad \hat{\Gamma} \vdash [v/x]x : \overrightarrow{\tau}' \setminus C'$$

Case $[y]$. This case trivially preserves types as $[v/x]y = y$ for all $x \neq y$ and the type of y is identical before and after substitution.

$$\hat{\Gamma},\ x : \overrightarrow{\tau}' \vdash y : \overrightarrow{\tau} \setminus C \qquad\qquad \hat{\Gamma} \vdash [v/x]y : \overrightarrow{\tau} \setminus C$$

The value cases are trivial. All other cases (abstraction, application, sequence, tuple, projection, tagged expressions, case, fix, arrow application, async, and advance) hold by trivial application of the expression's typing rule and the inductive hypothesis. □

Theorem 3 (Preservation of $\hat{\Gamma}$)**.** If $\Delta,\ \hat{e} \to \Delta',\ \hat{e}'$, Δ is well-formed, and $\hat{\Gamma} \vdash \hat{e} : \overrightarrow{\tau} \setminus C$, then $\exists \overrightarrow{\tau}' \setminus C'$ such that $\hat{\Gamma} \vdash \hat{e}' : \overrightarrow{\tau}' \setminus C'$ and one of the following conditions hold.

1. $\hat{e} = \langle e \rangle$,
2. $\hat{e} \neq \langle e \rangle$, $\overrightarrow{\tau} = \overrightarrow{\tau}'$ and $\mathit{closure}(C') \subseteq \mathit{closure}(C)$, or
3. $\hat{e} \neq \langle e \rangle$, $\overrightarrow{\tau} \neq \overrightarrow{\tau}'$ and $\mathit{closure}(C' \cup \{\overrightarrow{\tau}' \leq \overrightarrow{\tau}\}) \subseteq \mathit{closure}(C)$.

Proof. We prove by case analysis on $\hat{e}$.

Case $[\mathcal{E}[e_0]]$. $\Delta,\ e_0 \to \Delta',\ e_0'$ by the induction hypothesis and $\hat{e}' = \mathcal{E}[e_0']$ by rule (E-Congruence). It is straightforward to show by case analysis on the form of $\mathcal{E}$ that this case preserves types.

Case $[\lambda x.e_0\ v]$. By rule (E-App), $\hat{e}' = [v/x]e_0$. By rules (TA-App) and (TA-Abs), we have the following.

$$\frac{\hat{\Gamma},\ x : \vec{\tau_1} \vdash e_0 : \vec{\tau_2} \setminus C_1}{\hat{\Gamma} \vdash \lambda x.e_0 : \vec{\tau_1} \to \vec{\tau_2} \setminus C_1 \qquad \hat{\Gamma} \vdash v : \vec{\tau_3} \setminus C_2}{\hat{\Gamma} \vdash \lambda x.e_0\ v : \vec{\tau_2} \setminus C_1 \cup C_2 \cup \{\vec{\tau_3} \leq \vec{\tau_1}\}}$$

By rule (TA-Sub), we have the following.

$$\frac{\hat{\Gamma} \vdash v : \vec{\tau_3} \setminus C_2}{\hat{\Gamma} \vdash v : \vec{\tau_1} \setminus C_2 \cup \{\vec{\tau_3} \leq \vec{\tau_1}\}}$$

By the premise above and the substitution lemma, $\hat{\Gamma} \vdash [v/x]e_0 : \vec{\tau_2} \setminus \hat{C}$, where $C_1 \subseteq \hat{C} \subseteq C_1 \cup C_2 \cup \{\vec{\tau_3} \leq \vec{\tau_1}\}$.

Case $[v;\ e]$. By rule (E-Seq), $\hat{e}' = e$. By rules (TA-Seq), we have the following.

$$\frac{\hat{\Gamma} \vdash v : \top \setminus C_1 \qquad \hat{\Gamma} \vdash e : \top \setminus C_2}{\hat{\Gamma} \vdash v;\ e : \top \setminus C_1 \cup C_2}$$

As $C_2 \subseteq C_1 \cup C_2$, this reduction preserves types.

Case $[(v_i)^{i \in 1..j..n}[j]]$. By rule (E-Proj), $\hat{e}' = v_j$. By rule (TA-Proj), we have the following.

$$\frac{\hat{\Gamma} \vdash (v_i)^{i \in 1..n} : (\tau_i)^{i \in 1..n} \setminus C}{\hat{\Gamma} \vdash (v_i)^{i \in 1..n}[j] : \tau_j \setminus C}$$

By the premise above, $\hat{\Gamma} \vdash v_j : \tau_j \setminus C$, and both $\hat{e}$ and $\hat{e}'$ have identical types.

Case $[\textbf{\textit{case}}\ \ell_k(v)\ \textbf{\textit{of}}\ \{\ell_i(x_i) \Rightarrow e_i\}^{i \in 1..k..n}]$. By rule (E-Case), $\hat{e}' = [v/x_k]e_k$. By rule (TA-Case), we have the following.

$$\frac{\hat{\Gamma} \vdash \ell_k(v) : \langle \ell_k : \tau_k \rangle \setminus C_0 \qquad \hat{\Gamma},\ x_i : \tau_i \vdash e_i : \top \setminus C_i}{\hat{\Gamma} \vdash \textbf{case}\ \ell_k(v)\ \textbf{of}\ \{\ell_i(x_i) \Rightarrow e_i\}^{i \in 1..k..n} : \top \setminus C_0 \cup \bigcup C_i}$$

By deconstruction of the first premise above, we have $\hat{\Gamma} \vdash v : \tau_k \setminus C_0$. By the substitution lemma, we have $\hat{\Gamma} \vdash [v/x_k]e_k : \top \setminus \hat{C}$ where $C_k \subseteq \hat{C} \subseteq C_0 \cup C_k$. As $\hat{C} \subseteq C_0 \cup \bigcup C_i$, this reduction preserves types.

Case $[v_a \bullet (v,\ v_p,\ v_k,\ v_h)]$. By rule (E-Arrow-App), we have the following.

$$\hat{e}' = [v/x,\ v_p/p,\ v_k/k,\ v_h/h]e_0$$

where $v_a = \lambda x.\lambda p.\lambda k.\lambda h.e_0$. Then, by rule (TA-Arrow-App) and by repeated application of rule (TA-Abs), we have the following.

$$\frac{\hat{\Gamma} \vdash v_a : \tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top \setminus C_0 \qquad \hat{\Gamma} \vdash v : \tau_1 \setminus C_1 \qquad \hat{\Gamma} \vdash v_p : \tau_p \qquad \hat{\Gamma} \vdash v_k : \tau_2 \to \top \setminus C_2 \qquad \hat{\Gamma} \vdash v_h : \tau_3 \to \top \setminus C_3}{\hat{\Gamma} \vdash v_a \bullet (v,\ v_p,\ v_k,\ v_h) : \top \setminus C_0 \cup C_1 \cup C_2 \cup C_3}$$

$$\frac{\dfrac{\hat{\Gamma},\ x : \tau_1,\ p : \tau_p,\ k : \tau_2 \to \top,\ h : \tau_3 \to \top \vdash e_0 : \top \setminus C_0}{\vdots}}{\hat{\Gamma} \vdash \lambda x.\lambda p.\lambda k.\lambda h.e_0 : \top \setminus C_0}$$

By the premise above and the substitution lemma, we have the following where $C_0 \subseteq \hat{C} \subseteq C_0 \cup C_1 \cup C_2 \cup C_3$.

$$\hat{\Gamma} \vdash [v/x,\ v_p/p,\ v_k/k,\ v_h/h]e_0 : \top \setminus \hat{C}$$

As $\hat{C} \subseteq C_0 \cup C_1 \cup C_2 \cup C_3$, this reduction preserves types.

Case $[v_f\ v]$. By rule (E-Host-App), $\hat{e}' = v'$ where $(v_f\ v) \downarrow v'$. By rule (TA-Host-App), we have the following.

$$\frac{\texttt{Annot}_F(v_f) = \tau_1 \to \tau_2 \setminus (C_0,\ E) \qquad \hat{\Gamma} \vdash v : \tau_1 \setminus C_1}{\hat{\Gamma} \vdash (v_f\ v) : \langle succ : \tau_2,\ fail : \alpha \rangle \setminus C_0 \cup C_1 \cup \{\tau \leq \alpha \mid \tau \in E\}}$$

By rule (E-Host-App), a runtime check ensures that

$$\emptyset \vdash v' : \langle succ : \tau_2,\ fail : \tau_3 \rangle$$

where $\tau_3 \in E$ or ($E = \emptyset$ and $\tau_3 \in \top$). As the base types are not equivalent, we must show that the following relation holds.

$$\begin{aligned} & closure(\{\langle succ : \tau_2,\ fail : \tau_3 \rangle \leq \langle succ : \tau_2,\ fail : \alpha \rangle\}) \\ \subseteq\ & closure(C_0 \cup C_1 \cup \{\tau \leq \alpha \mid \tau \in E\}) \end{aligned}$$

If $E = \emptyset$, then we have $\{\top \leq \alpha\} \subseteq C_0 \cup C_1$ - as α is unbounded, the left hand side is equivalent to the trivial constraint $\top \leq \top$, which can be discarded. If $E \neq \emptyset$, then we have $\{\tau_3 \leq \alpha\} \subseteq C_0 \cup C_1 \cup \{\tau_3 \leq \alpha\} \cup \{\tau \leq \alpha \mid \tau \in E \setminus \{\tau_3\}\}$.

Case $[\texttt{fix}(\lambda\omega.e_0)]$. By rule (E-Fix), $\hat{e}' = [\texttt{fix}(\lambda\omega.e_0)/\omega]e_0$. By rule (TA-Fix), we have the following.

$$\frac{\mathrm{T} \equiv \tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top \qquad \hat{\Gamma},\ \omega : \mathrm{T} \vdash e_0 : \mathrm{T} \setminus C}{\hat{\Gamma} \vdash \texttt{fix}(\lambda\omega.e_0) : \mathrm{T} \setminus C}$$

By the second premise and the substitution lemma, we have the following.

$$\hat{\Gamma} \vdash [\texttt{fix}(\lambda\omega.e_0)/\omega]e_0 : \mathrm{T} \setminus C$$

As both $\hat{e}$ and $\hat{e}'$ have identical types.

Case $[\texttt{async}\ v_e\ v_p\ \lambda x.e_0]$. By rule (E-Async), $e' = ()$. By rule (TA-Async), we have the following.

$$\frac{\hat{\Gamma} \vdash v_e : \langle \mathit{succ} : \tau_1,\ \mathit{fail} : \tau_2 \rangle \setminus C_1 \qquad \hat{\Gamma} \vdash v_p : \tau_p \qquad \hat{\Gamma} \vdash \lambda x.e_0 : \tau_3 \to \top \setminus C_2}{\hat{\Gamma} \vdash \texttt{async}\ v_e\ e_p\ \lambda x.e_0 : \top \setminus C_1 \cup C_2 \cup \{\langle \mathit{succ} : \tau_1,\ \mathit{fail} : \tau_2 \rangle \leq \tau_3\}}$$

By rule (TA-Unit), we have $\hat{\Gamma} \vdash () : \top$ and $\emptyset \subseteq C$.

Case $[\langle v \rangle]$. In this case, we assume an event $v_e \in \Delta$ has completed. By rule (E-Event), $v_e \mapsto (v_p,\ \lambda x.e_0) \in \Delta$ and $e' = [\texttt{Resp}(v_e)/x]e_0$. Because Δ is well-formed and by rule (TA-Sub), we have the following.

$$\frac{\emptyset \vdash v_e : \langle \mathit{succ} : \tau_1,\ \mathit{fail} : \tau_2 \rangle \setminus C_1}{\emptyset \vdash v_e : \tau_3 \setminus C_1 \cup \{\langle \mathit{succ} : \tau_1,\ \mathit{fail} : \tau_2 \rangle \leq \tau_3\}} \qquad \emptyset \vdash \lambda x.e : \tau_3 \to \top \setminus C_2$$

By rule (TA-Abs), we have $\emptyset,\ x : \tau_3 \vdash \hat{e} : \top \setminus C_2$ and by the substitution lemma, we have the following, where $C_2 \subseteq \hat{C} \subseteq C_1 \cup C_2 \cup \{\langle \mathit{succ} : \tau_1,\ \mathit{fail} : \tau_2 \rangle \leq \tau_3\}$.

$$\emptyset \vdash [\texttt{Resp}(v_e)/x]e : \top \setminus \hat{C}$$

As Δ is well-formed, this constraint set above is consistent and $\hat{e}'$ is well-typed.

Case $[\texttt{adv}\ (P_i^j :: v_p)]$. By rule (E-Advance), $\hat{e}' = \texttt{adv}\ v_p$. By rules (TA-Advance) and (TA-Prog), we have the following.

$$\frac{\dfrac{P_i^j \in \{P_i^j,\ Q_i\} \qquad \hat{\Gamma} \vdash v_p : \tau_p}{\hat{\Gamma} \vdash P_i^j :: v_p : \tau_p}}{\hat{\Gamma} \vdash \texttt{adv}\ P_i^j :: v_p : \top} \qquad\qquad \frac{\Gamma \vdash v_p : \tau_p}{\hat{\Gamma} \vdash \texttt{adv}\ v_p : \top}$$

Both $\hat{e}$ and $\hat{e}'$ have identical types, and this reduction preserves types.

Case $[\texttt{adv}\ e_p]$ *where* $e_p \neq (P_i^j :: v_p)$. e_p has the form $Q_i^j :: v_p$ or ϵ. $\hat{e}' = \texttt{()}$ by respective rules (E-Advance-Quiet) and (E-Advance-Empty) for each case, respectively. By rules (TA-Advance), (TA-Prog), and (TA-Prog-Empty), we have the following.

$$\frac{\dfrac{Q_i \in \{P_i^j,\ Q_i\} \qquad \hat{\Gamma} \vdash v_p : \tau_p}{\hat{\Gamma} \vdash Q_i :: v_p : \tau_p}}{\hat{\Gamma} \vdash \texttt{adv}\ Q_i :: v_p : \top} \qquad\qquad \frac{\hat{\Gamma} \vdash \epsilon : \tau_p}{\hat{\Gamma} \vdash \texttt{adv}\ \epsilon : \top}$$

By rule (TA-Unit), $\hat{\Gamma} \vdash \texttt{()} : \top$ and both $\hat{e}$ and $\hat{e}'$ have identical types.

This completes case analysis on $\hat{e}$, and the proof. □

Appendix E. Proof of Theorem 4 (See page 40)

Theorem 4 (Progress)**.** If $\emptyset \vdash \hat{e} : \overrightarrow{\tau} \setminus C$ and Δ is well-formed, then either one of the following conditions hold.

1. $\hat{e} = v$,
2. $\hat{e} = \langle v \rangle$ and $\Delta = \emptyset$,
3. $\exists \Delta',\ \hat{e}'$ such that $\Delta,\ \hat{e} \to \Delta',\ \hat{e}'$, or
4. a typing premise fails in rule (E-Host-App) or (E-Event).

Proof. We prove by case analysis on $\hat{e}$.

Case $[\mathcal{E}[e_0]]$. In this case, we assume e_0 is not a value. Then, by the induction hypothesis, either $\Delta,\ e_0 \to, \Delta',\ e_0'$ and $\hat{e}' = \mathcal{E}[e_0']$, or a typing premise fails while reducing the subexpression e_0.

Case $[v_1\ v_2]$. By inversion of types, $v_1 = \lambda x.e_0$ and $\hat{e}' = [v_2/x]e_0$ by rule (E-App).

Case $[v;\ e]$. By rule (E-Seq), $\hat{e}' = e$.

Case $[v[j]]$. By rule (TA-Proj), $\hat{\Gamma} \vdash v : (\tau_i)^{i \in 1..n}$ and $1 \leq j \leq n$. By inversion of types, $v = (v_i)^{i \in 1..n}$ and $\hat{e}' = v_j$ by rule (E-Proj).

Case $[\textbf{\textit{case}}\ v\ \textbf{\textit{of}}\ \{\ell_i(x_i) \Rightarrow e_i\}^{i \in 1..n}]$. By rule (TA-Case), we have the following.

$$\emptyset \vdash v : \langle \ell_i : \tau_i \rangle^{i \in 1..n} \setminus C$$

By inversion of types, $e_0 = \ell_k(v)$ for some $1 \leq k \leq n$ and $\hat{e}' = [v/x_k]e_k$ by rule (E-Case).

Case $[v_a\ \bullet\ (v_0,\ v_p,\ v_k,\ v_h)]$. By rule (TA-Arrow-App), we have the following.

$$\hat{\Gamma} \vdash v_a : \tau_1 \to \tau_p \to (\tau_2 \to \top) \to (\tau_3 \to \top) \to \top \setminus C$$

By inversion of types, $e_a = \lambda x.\lambda p.\lambda k.\lambda h.e_1$ and $\hat{e}' = [v_0/x,\ v_p/p,\ v_k/k,\ v_h/h]e_1$ by rule (E-Arrow-App).

Case $[v_f\ v]$. By rule (TA-Host-App), $\texttt{Annot}_F(v_f) = \tau_1 \to \tau_2 \setminus (C,\ E)$ and $(v_f\ v) \downarrow v'$ by (E-Host-App). If $\emptyset \vdash v' : \langle succ : \tau_2 \rangle$ or $\emptyset \vdash v' : \langle fail : \tau_3 \rangle$ such that either $\tau_3 \in E$, or $E = \emptyset$ and $\tau_3 = \top$, then $\hat{e}' = v'$. Otherwise, a typing premise failed and v_f returned an value inconsistent with its annotation.

Case $[\texttt{fix}(\lambda\omega.e_a)]$. By rule (E-Fix), $e' = [\texttt{fix}(\lambda\omega.e)/\omega]e_a$.

Case $[\texttt{async}\ v_e\ v_p\ v_k]$. By rule (E-Async), $\hat{e}' = \texttt{()}$ and $\Delta' \supset \Delta$.

Case $[\langle v \rangle]$. If $\Delta = \emptyset$, then no further reductions are possible. Otherwise, $v_e \mapsto (v_p,\ \lambda x.e_0) \in \Delta$. Let $\tau \setminus C$ and E be the type and error set annotation of v_e, respectively. If $\emptyset \vdash \texttt{Resp}(v_e) : \langle succ : \tau_2 \rangle$ or $\emptyset \vdash \texttt{Resp}(v_3) : \langle fail : \tau_3 \rangle$ such that either $\tau_3 \in E$, or $E = \emptyset$ and $\tau_3 = \top$, then $\hat{e}' = [\texttt{Resp}(v_e)/x]e_0$ and $\Delta' \subset \Delta$ by rule (E-Event). Otherwise, a typing premise failed and v_e had produced a value inconsistent with its annotation.

Case $[\texttt{adv}\ v_p]$. If v_p is ϵ or $Q_i :: v'_p$, then $\hat{e}' = \texttt{()}$ by rule (E-Advance-Empty) and rule (E-Advance-Quiet), respectively. If v_p is $P_i^j :: v'_p$, then $\hat{e}' = v_p$, and $\Delta' \subseteq \Delta$ by rule (E-Advance).

This completes case analysis on $\hat{e}$, and the proof. □