Typing and Semantics of Asynchronous Arrows in JavaScript

Eric Fritz, Tian Zhao

University of Wisconsin – Milwaukee

Abstract

Asynchronous programs in JavaScript using callbacks and promises are difficult to write correctly. Many programs have subtle errors due to the unwanted interaction of event handlers. To fix such errors, the programmer is burdened with explicit registration and de-registration of event handlers. This produces fragile code which is difficult to read and maintain.

Arrows, a generalization of monads, are an elegant solution to asynchronous program composition. In this paper, we present the semantics of an arrow-based DSL in JavaScript which can encode asynchronous programs as a state machine where edge transitions are triggered by external events. To ensure that arrows are composed correctly, we provide an optional type checker that reports errors before the machine begins execution.

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 to JavaScript 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 (see Section 2.1 for a concrete comparison between callbacks and promises). While promises can help avoid nested callbacks, promises alone cannot express the same semantics as callbacks without relying on external state and manual synchronization. For example, `Promise.race([p1, p2])` causes the promises `p1` and `p2` to execute concurrently such that the result of the race is that of the *faster* promise. However, both `p1` and `p2` run to completion, which can be problematic if the slower one has undesirable side effect such as loading a large file. To cancel the slower promise, `p1` and `p2` must coordinate through shared state to indicate completion of the first-to-resolve promise, which is not an improvement over callbacks.

In addition, promises cannot encode general recursion without relying on recursive function calls. This diminishes the clarity of promise-based programs as readers have to switch between higher-level constructs of promises to lower-level function calls when trying to understand the asynchronous dataflow. We address these problems with a different abstraction for asynchronous composition, arrows, introduced in Section 1.1.

Both callback and promise compositions are error-prone to compose as JavaScript lacks the ability to determine *statically* when a callback sequence or promise chain is illegal. Such compositions often crash or lead to subtle behavioral issues once invoked. Frustratingly, the source location which displays incorrect behavior is often completely independent of the source location of the root cause, making the associated stack trace less than helpful. These errors force the developer to trace the 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.

```
1  function lookup(key) {
2      return new Promise((resolve, reject) => {
3          if (key in cache) { resolve(cache[key]); }
4          else              { reject(key); }
5      });
6  }
7
8  let resp = lookup(key).catch(key => makeRequest(key).then(resp => {
9      cache[key] = resp;
10 }));
11
12 resp.then(items => handleFirst(items[0]));
```

Figure 1: A (malformed) program using promises to cache remote server responses.

The code in Figure 1 demonstrates such an illegal composition using promises. The function `lookup` returns a promise that resolves with a value from a cache (if it exists) or rejects with the key. When this promise rejects, the error handling function is invoked in order to make a remote server request which populates the cache with the result. Although the promise chain is sequenced correctly, it contains an error which may not be immediately obvious to the reader. Because the function that populates the cache returns no value, the promise `resp` will resolve to the value `undefined`. In this example, the illegal composition results in an index out of bounds exception on line 12, when the actual error is in the construction of the promise chain. One solution to this problem requires that the function that populates the cache also returns the cached value. Another solution requires that the promise returned by `makeRequest` be returned without modification. In this solution, the function populating the cache would be sequenced to the promise internally for side-effect only.

We address this composition issue with respect to arrows with an optional type checker to detect illegal use before execution, discussed in detail in Sec-

tion 4. Section 2 Figure 6 presents a similar (but typed) program using arrows.

1.1. Arrows

In this paper, we introduce an arrows library for composing asynchronous JavaScript programs. Arrows are easier to use than callbacks and promises as they enable thread-like semantics for event-based programming with the ability to properly cancel an asynchronous computation (rather than just ignoring its results as promises do).

Arrows [1] are a generalization of *monads* [2], which enable flexible composition of actions on a particular type of data. A monad **m** includes a `return` operator that turns a value of type a into a monad of type **m** a and a `bind` operator that combines a monad of type **m** a and a function $a \rightarrow$ **m** b to return a monad of type **m** b. Monads are considered a flexible design for combinator libraries. For example, the *Maybe* monad can be used to define combinators so that programs composed with the combinators can handle errors implicitly through the `bind` operator.

While Monad enables flexible combinator libraries, it does have some limitations. For instance, Swierstra and Duponcheel [3] defined efficient parser combinators, with the type **Parser** s a that has a static and a dynamic component, to match strings from LL(1) grammars. The static component allows a *choice* combinator to make a decision immediately on which parser to invoke. Similar parsers that do not consider lookahead must retain the input (which has the potential to be very large) while invoking one side of the choice as they cannot determine whether a parser will match or fail ahead of time.

Unfortunately, as John Hughes showed [1], it is not possible to define the `bind` operator for their parser combinators, which has the following type.

$$\textbf{Parser}\ s\ a \rightarrow (a \rightarrow \textbf{Parser}\ s\ b) \rightarrow \textbf{Parser}\ s\ b$$

The static component of the result depends on the static components of *both* inputs, but the second parser's static component is hidden behind a function requiring a value of type a.

Through a reformulation of the required type signatures, arrows generalize over monads so that the *static* properties of the composed arrows are readily available. Arrows include a `return` operator (also called `lift`) that turns a function of type $b \rightarrow c$ into an arrow of the type $\mathbf{a}\ b\ c$, where $\mathbf{a}$ is a type constructor applied to the parameters b and c, a `seq` operator for sequencing arrows of type $\mathbf{a}\ b\ c$ and $\mathbf{a}\ c\ d$ to result in an arrow of type $\mathbf{a}\ b\ d$, and a `first` operator, which takes an arrow of type $\mathbf{a}\ b\ c$ and convert it to an arrow of type $\mathbf{a}\ (b,\ d)\ (c,\ d)$ so that value of type d can be piped through the arrow unchanged.

Our design is motivated by Arrowlets [4], which has demonstrated an elegant solution to composing callback functions by wrapping them in opaque units of execution (arrows) using continuation functions. Arrows are suitable for composing continuations that represent event callbacks in JavaScript, where the progress of the event callbacks must be monitored. Arrows, unlike promises and callbacks, have a distinct *composition* phase and an *execution* phase. An arrow is composed in-full before its invocation is kicked off by calling its `run` method. The portion of the program which executes before the `run` method is called is referred to as composition phase, and the remaining portion of the program is referred to as execution phase.

While both phases occur at runtime, the actual computation including the asynchronous code only starts at the execution phase. Figure 2, which presents the construction of an arrow encoding the game *Whack-A-Mole*, shows why composition is a runtime concept rather than a *static* (compilation-time) concept. In this example, nine arrows are created, one controlling the timing of each mole (details omitted), which runs concurrently. The difference between the nine arrows is only the element on which they operate, which has been factored out as a parameter to the arrow factory function. If arrow composition was a static concept, the host language cannot be meaningfully leveraged in this way.

Because of this separation, it is possible to detect errors after an arrow has been composed but before its actual execution starts. To this end, we have developed an optional type checker which infers and attaches a type to every

```
function popup(selector) {
    return Arrow.seq([/* omitted */]);
}

var moles = [];
for (var i = 1; i <= 9; i++) {
    moles.push(popup('#mole' + i));
}

Arrow.all(moles);
```

Figure 2: An arrow *factory* function showing why composition phase is not static.

arrow during the composition phase 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 (in the form of a JavaScript comment) to functions which are *lifted* into arrows.

Arrows and promises share a large enough core set of semantics that our typing rules can also apply almost directly to a promise library. Unfortunately, promises evaluate as soon as they are defined and there is no separation between the definition of a promise and its evaluation. An equivalent type checker for promises would either require a static pre-processing step (which would require not only knowledge of promise semantics, but the semantics of JavaScript as well), or would immediately devolve into inspecting its argument types during runtime. Such a type system would not be beneficial, as it would not detect composition errors before the promise chain executes.

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 statically, it does find errors prior to the *arrow execution phase*. More precisely, the type checker ensures that the arrows are composed correctly based on the arrow's types so that the argument passed to a lifted function is always compatible with the corresponding parameter type of the function. This technique effectively moves the source of errors from the point where an error may be ob-

served to the point where an erroneous composition occurs. In a sense, the type checking is static relative to the arrow execution. If typing errors are detected, users can quickly identify the illegal arrow compositions and revise accordingly. Since this process does not require actual asynchronous computation, the development of arrows can often be less time consuming than that of promises. The type checker may also be disabled, returning the program to the original runtime semantics without dynamic type checks.

The type checker infers the types of primitive arrows and their compositions. The domain and the range of arrow types are limited to primitive types, type variables, and immutable tuple, record, array, and variant types. This set of types is sufficient for the purpose of expressing interesting asynchronous behavior so that is not necessary to consider more complex issues such as mutable object fields, prototypes, and higher order functions. Since we require type annotations for each *lifted* function and perform (optional) run-time checks to ensure that the function return value matches the annotated type, we do not need to perform type-inference within the body of the lifted function.

1.2. 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 the asynchronous arrows library and its relation to JavaScript's single-threaded event loop.

The remainder of this paper is organized as follows. Section 2 provides a motivation for using arrows as a tool for asynchronous composition, and a motivation for inferring types of the asynchronous machine produced by arrow composition. 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 development time and runtime costs of the 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. Motivation

In this section, we provide motivation for using arrows as a utility to compose asynchronous functions as well as a motivation for annotating such compositions with types.

In the following, we illustrate compositions with code snippets using the arrows library. The details of the methods are explained in detail 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, `any` executes two arrows in parallel and cancels the execution of the *slower* of the two, `fix` allows an arrow to reference itself recursively, `catch` executes the second arrow when an exception occurs in the first, and `on` blocks until an event occurs on an element. An arrow begins execution only after its `run` method is invoked. We encourage readers to reference Section 3 in order to understand the complete composition.

2.1. Why Arrows?

To illustrate the utility of arrows as a composition utility, we give an implementation of a small program using only callbacks, using promises, and using arrows. Each implementation assumes the existence of the function `makeURL`, which returns a path on a remote server, and the function `handle`, which processes results from a remote server.

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

The program waits for the user to click an element with the document ID *fire*. Then, an Ajax request will fire after a one-second delay. Once the remote server responds, the results are passed to the `handle` function. Once the results are handled, the program halts. If the user clicks the element a second time before the results are handled, the results of the timer and Ajax request are discarded and a new timer begins.

The callback and promise solutions use the jQuery method `one`, which binds an event handler to an event which fires only once (unlike the jQuery method `on`, which will not de-register the handler after the first invocation).

```
function inner() {
    var clicked = false;
    var responded = false;

    setTimeout(() => {
        if (clicked) return;
        $.ajax({
            'url': makeURL(),
            'success': response => {
                if (clicked) return;
                responded = true;
                handle(response);
            }
        });
    }, 1000);

    $('#fire').one('click', () => {
        if (responded) return;
        clicked = true;
        inner();
    });
}

$('#fire').one('click', inner);
```

Figure 3: A solution using only callbacks.

Callbacks. Figure 3 shows a complete implementation of this program using only callbacks. We define the method `inner` as the computation to occur after

a click event. This method is called once from outside in order to kick-off the computation and from inside on additional click events. The flag `clicked` is set to true on an additional click event and will prevent the results of the *previous* Ajax call from being handled. If an Ajax response is received and the `clicked` value has not been set to true, then the results are sent to the `handle` method and the program halts. The flag `responded` is set to true directly before `handle` is called. Additional click handlers will no-op if the `responded` flag has been set to true. Both flags are necessary in order to prevent unwanted behavior from occurring – otherwise, `handle` maybe called twice, or clicks may be registered after the program has ended.

```
1  function inner() {
2      var p1 = new Promise((resolve, reject) =>
3          setTimeout(resolve, 1000)).then(() => $.ajax({ 'url': makeURL() }));
4
5      var p2 = new Promise((resolve, reject) =>
6          $('#fire').one('click', reject));
7
8      Promise.race([p1, p2]).then(handle, inner);
9  }
10
11 $('#fire').one('click', inner);
```

Figure 4: A solution using promises.

Promises. Figure 4 shows a complete implementation of this program using promises. Promise `p1` begins a timer and resolves with the Ajax response once it is received from the remote server and promise `p2` rejects when an additional click is detected. Both promises *race* so that the first resolved value is acted on and the result of the slower promise is silently discarded. If the result of the race is a rejection, then the `inner` method is called recursively; otherwise, the Ajax response is sent to the `handle` function and the program halts.

Arrows. Figure 5 shows a complete implementation of this program using arrows. The `AjaxArrow` constructed on line 7 describes a computation that makes

```
Arrow.fix(a => Arrow.seq([new ElemArrow('#fire'), Arrow.on('click',
    Arrow.any([
        a,
        Arrow.seq([
            Arrow.noemit(Arrow.seq([
                new DelayArrow(1000),
                new AjaxArrow(() => { 'url': makeURL() })
            ])),
            handle
        ])
    ])
)])).run();
```

Figure 5: A solution using Arrows.

a request to the remote server and yields the response. A reference to the recursive parameter `a` within the `fix` call effectively replays the arrow execution from the beginning. The arrow is immediately executed after composition via the `run` method.

The arrow begins by retrieving an element with the document ID *fire* and waits for a click event. Once the event occurs, the arrow re-invokes itself (line 3) to register another click handler. Control is then yielded to the arrow defined on lines 4-10 which sequences a timeout, the ajax arrow, and the `handle` function. The `any` combinator will wait for either the click event or the Ajax response to occur. Once one event occurs, the event handlers of the other branch are de-registered and the *winning* branch continues execution.

Callback Discussion. The solution using promises and the solution using arrows are both higher-level than the solution using only callbacks. In order to ensure that only one click handler and only one Ajax request is in-flight at a time, the solution using only callbacks resorts to using function-local flags which are used to determine if an event should be ignored. An alternate solution would keep a reference to the timer and the `XMLHttpRequest` object so that the `clearTimeout` function and the request's `abort` method could be used to cancel event handlers eagerly instead of lazily – this implementation would remain as explicit as the

one given in Figure 3. The solution using promises and the solution using arrows, on the other hand, discards the branch containing the callback for the *slower* event implicitly. The semantics of `Promise.race` and `Arrow.any` already encode this behavior in a reusable way.

Promise Discussion. The promise and arrow solutions are similar, but the arrow solution is more efficient. The `Promise.race` method resolves to the faster (first to resolve) promise, but the remaining promises remain active. On the other hand, the `Arrow.any` combinator de-registers the event callbacks from the slower arrows once one arrow makes sufficient progress. This effectively cancels branches of computation whose results will not be used.

While this difference in semantics is a matter of superfluous (but silent) remote requests in this program, it is a matter of correctness in others. Suppose that two concurrent branches of computation are both waiting on an event to proceed – if the computation remaining on these branches is user-observable, it may be incorrect to allow one to continue execution while discarding its result. Unlike arrows, ES6 Promises do not offer a mechanism for cancellation. Section 3.2 describes in greater detail the ability for arrows to offer semantics equivalent to `Promise.race`.

2.2. Why a Type System?

To illustrate the utility of our type inference tool, consider the example in Figure 6 which wraps an arrow with a cache. The function `makeCached` builds an arrow that first checks for the existence of a key in a globally scoped object. If this key exists, then the arrow returns this value immediately. Otherwise, the key does not exist in the cache and the arrow is invoked to calculate a value which is placed in the cache before returning.

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 any value and may return any value. The arrow resulting from `makeCached` first invokes the lifted function `lookup`, which will either return a value from

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

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

function makeCached(arr) {
    return Arrow.try(lookup, id, Arrow.seq([
        arr.carry(),
        store.remember(),
        new NthArrow(2)
    ]));
};
```

Figure 6: An example arrow composition caches results of a wrapped arrow. The type annotations have the form of $\tau_1 \rightsquigarrow \tau_2 \setminus (C, E)$, where τ_1 and τ_2 are input and output types, C is a set of type constraints on τ_1 and τ_2, and E is a set of types for possible exceptions. We omit (C, E) when there is no type constraints or exceptions. α, β, γ, and δ range over type variables and $\top$ indicates no useful return value.

the cache on cache hit or throw the key on cache miss. On cache miss, the exception value is caught and fed into `arr` and execution continues on the error-handling branch.

This solution is general in the sense that no concrete type is listed for either the keys or values within the cache – these types are inferred based on the input and output type of the arrow being wrapped.

The *derived* combinator `carry` wraps an arrow so that its input is returned along with the output (e.g. it converts an arrow of type $\alpha \rightsquigarrow \beta$ to $\alpha \rightsquigarrow (\alpha,\ \beta)$). The derived combinator `remember` wraps an arrow so that its input is returned in place of its output and the arrow is executed simply for side-effects (e.g. it converts an arrow of type $\alpha \rightsquigarrow \beta$ to $\alpha \rightsquigarrow \alpha$).

Notice that the type of `store` requires a key of type γ and a value of type

δ as input, but `arr` returns only a value of some unknown type. To give `store` the correct input, we need to remember the input of `arr`. Similarly, `store` does not return any usable value, so we must remember its output as well in order to extract the value placed in the cache.

If, instead, `arr` and `store` were sequenced directly, e.g. if the `carry` combinator was incorrectly omitted, it would result in a (rightful) type-clash. Such a composition would calculate a value but store `undefined` in the cache under the wrong key, resulting in a likely cryptic error later in the execution of the program when `undefined` is read from the cache. If type checking was enabled, the direct sequencing of `arr` and `store` would not type check, giving the developer immediate feedback that the arrows are ill-composed. This composition-time failure has two advantages over the runtime failure, as follows.

1. The *location* of the error now occurs at the root cause (where the illegal composition occurs) rather than at the first symptom.

2. The *text* of the error message contains the inferred types of the arrows being composed as well as the reason the composition is not type-safe.

As a concrete example, if `arr` has type $\mathsf{String} \rightsquigarrow [\mathsf{String}]$ then the following error message is given at composition time. This gives the user precisely the types which are incompatible.

Cannot seq arrow: Inconsistent constraints $\{[\mathsf{String}] \leq (\gamma,\ \delta)\}$

It is worth noting that the symptom of such an error, without type checking, can occur arbitrarily far away from the sequencing error. Such symptoms may only be observable under certain conditions (after a specific sequence of interactions) and in subtle ways (not causing a crash, but incorrect values). Finding such errors before they occur is certainly beneficial during development.

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. An arrow may receive a value as an argument as it begins execution. An arrow may also produce a value, but because an arrow may execute in an asynchronous manner, this value may only be consumed by another arrow.

We embed a typed domain-specific language based on arrow operations in 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 an arrow back into the host language.

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 recursive. The design and implementation of the library is heavily inspired by both Arrowlets [4] and ES6 Promises. We have, however, made a few major interface changes which are discussed in detail in Section 8.

Below, we define some concepts necessary for the discussion of arrows which are executed concurrently (specifically, in relation to the `any` and `noemit` combinators). Each arrow described below will explicitly state the location of its *async points*, if any, and whether or not it is *asynchronous*.

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. A running arrow may be canceled *only* at an async point, as cancellation is effectively the de-registration of its active event handlers so that it never resumes execution after an external event.

Definition 2 (Progress Event)**.** An arrow may emit a **progress event** if it successfully resumes execution after blocking at an async point. Listeners can subscribe to these events to know when an arrow is making progress. These events may be explicitly suppressed (as discussed in Section 3.2).

Definition 3 (Asynchronicity)**.** We say an arrow is **asynchronous** if it contains at least one async point on every possible execution path through the arrow. If an arrow contains an async point which may be avoided at runtime (e.g. via the `try` combinator), then it is not considered asynchronous.

3.1. Constructors

We provide seven arrow constructors (`lift`, `delay`, `domelem`, `domevent`, `ajax`, `nth`, and `split`) detailed below. These constructors were chosen to form an interesting and practical core on which our formalism is based and is by no means an exhaustive list of possible constructors. Additional constructors are likely to closely resemble a constructor already described here – for example, a `query` constructor, closely resembling the `ajax` constructor, might create an arrow running on the server-side which runs a query on a database.

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 7.

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

Figure 7: An example of a function *lifted* into an arrow.

A lifted arrow is synchronous, and we furthermore assume the body of a lifted function executes in a synchronous manner as any asynchronous behavior

within the lifted function is doing so outside of the arrow execution machine. This can be well-defined behavior, but these events will not produce observable async points, and a value produced asynchronously cannot be injected back into the 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.

Delay. The delay arrow, denoted `delay`(d), passes along its own input, unmodified, after d milliseconds pass. This arrow is asynchronous and emits an async point immediately after its internal timer fires.

Document 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.

Document 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 and emits an async point immediately after the event occurs.

In implementation, 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. This arrow is asynchronous and emits an async point immediately after it receives a response from the 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

```
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 8: An example of an arrow which asynchronously fetches data form a remote server.

server to ensure the shape of the data matches the annotated type. A concrete example is given in Figure 8.

Nth. The nth arrow, denoted `nth`(n), takes a tuple of *at least* n elements as input and extracts its nth element. Figure 9 demonstrates the dataflow of this arrow.

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. Figure 9 demonstrates the dataflow of this arrow.

This arrow precludes aliasing by creating n (deep) clones of the value v. This avoids problems with mutable references to values held by concurrently executing arrows. Note that while cloning is deep, it does not replicate values that are treated as atomic by our types such as events.

Cloning is essential to type safety of arrows where the argument passed to an arrow must have a type compatible to the arrow's input type. Without cloning, one branch of execution can mutate a value which is also used on another branch by reference (e.g. re-assigning or removing the fields of an object). This may make the value subsequently incompatible with the other branch's type.

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 runtime value (the number of branches of a split)

Figure 9: Dataflow diagrams for `split` and `nth` arrows.

and cannot be annotated statically with an accurate, fixed-size type.

3.2. Combinators

We provide five arrow combinators (`seq`, `all`, `try`, `any`, and `noemit`), 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. This set of combinators is not necessarily exhaustive, but is large enough that many interesting additional combinators can be *derived* (e.g. `carry` and `remember`) from this core set.

Seq. The sequence combinator, denoted $\texttt{seq}(a_1, \ldots, a_n)$, 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. The dataflow of the resulting arrow is demonstrated in Figure 10.

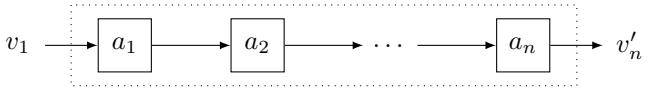

Figure 10: Dataflow diagram for the `seq` combinator.

This combinator is asynchronous if *any* arrow a_i is asynchronous. Each async point of arrow a_i is also an async point of the combinator.

This combinator generalizes the binary combinator $\ggg$ in the arrow calculus [1], typed as follows.

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

All. The all combinator, denoted $\texttt{all}(a_1, \ldots, a_n)$, composes n arrows that execute concurrently. This combinator begins executing each arrow, in order, in a synchronous loop. Once arrow a_i completes or reaches an async point, arrow a_{i+1} immediately begins execution. Once all arrows have been started, they may progress through their execution in any order until they all complete, at which point the combinator completes. The dataflow of the resulting arrow is demonstrated in Figure 11.

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.

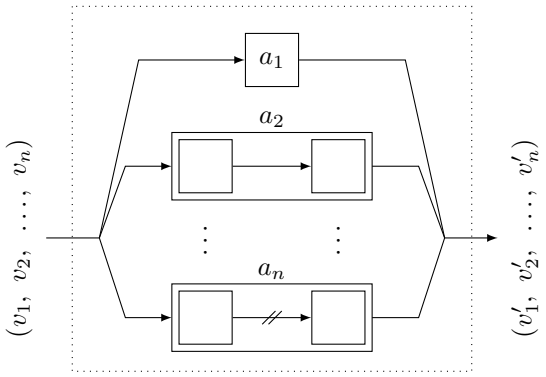

Figure 11: Dataflow diagram for the `all` combinator over arrows a_1 through a_n. The argument arrows can be simple or a complex result of other combinators (a_1 and a_2, respectively), and can be synchronous or asynchronous (a_2 and a_n, respectively).

This combinator is asynchronous if *any* arrow a_i is asynchronous. Each async point of arrow a_i is also an async point of the combinator.

We can construct a combinator equivalent to the unary combinator *first*, typed as follows, in the arrow calculus [1] using this combinator and an identity arrow *id* as follows.

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

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

Try. The try combinator, denoted $\texttt{try}(a, a_s, a_f)$, attempts to execute the *protected arrow* 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 the result of arrow a_f, depending on which one executed at runtime. The dataflow of the resulting arrow is demonstrated in Figure 12.

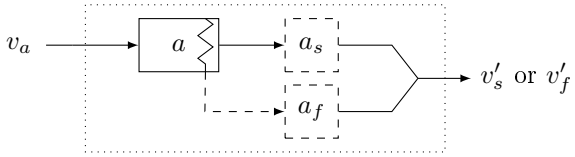

Figure 12: Dataflow diagram for the `try` combinator.

This combinator is asynchronous if every execution path contains an async point, regardless of where an exception may occur within the execution of a. We claim that the combinator is asynchronous if both arrow a_f and either arrow a or arrow a_s is asynchronous. This is an under-approximation which may tag some arrows whose execution always hit an async point as synchronous, but is a simple definition that works well in practice.

Promise's `then` and `catch` methods can be encoded by the `try` combinator. The statement $p.\texttt{then}(\textit{resolve})$ executes p and then the callback *resolve* on successful execution. The statement $p.\texttt{catch}(\textit{reject})$ executes p and, if an error occurs, calls *reject* with the error as input. The statement $p.\texttt{then}(\textit{resolve}, \textit{reject})$ 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 Figure 6 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 $\texttt{any}(a_1, \ldots, a_n)$, 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_*. The dataflow of the resulting arrow is demonstrated in Figure 13.

This combinator is asynchronous and will emit the *first* async point of each arrow a_i, then all remaining async points for arrow a_*.

The semantics of this combinator is ill-defined if one (or all) of the arrows is synchronous – a synchronous arrow would run to completion without giving its sibling arrows a chance to run due to the single-threaded nature of the event-loop. Therefore, this combinator enforces the constraint that all n arrows

are asynchronous during construction. This check can be easily performed at composition time.

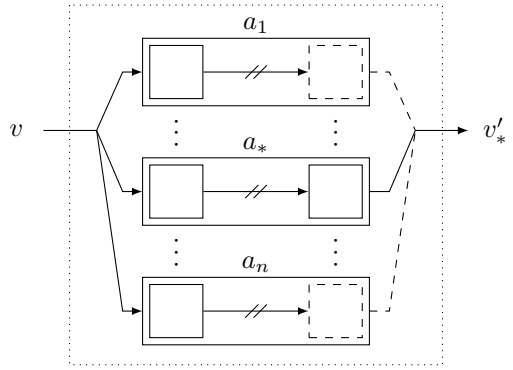

Figure 13: Dataflow diagram for the `any` combinator. Notice that all arrows have an async point.

Similar to the behavior of the `split` constructor, this creates n clones of the input value v before passing copies to the child arrows.

The result of this combinator differs from the result of the `Promise.race` method. In particular, 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. In the following example, the *task* arrow runs after a *window* second timeout only if the user does not click the element with the *cancel* selector within this period – the progress of either the `delay` or the `domevent` arrows are enough to determine whether or not *task* should run.

`any`(`seq`(`delay`(*window*), *task*), `seq`(`domelem`(*cancel*), `domevent`(*click*)))

A similar implementation using `Promise.race` will *resolve* when the click event occurs (regardless if it's within the initial *window* second period), or when *task* completes (when no click event occurs). Because a promise cannot be canceled, *task* will always run. This may be detrimental if it behaves in a user-observable manner.

NoEmit. The no-emit combinator, denoted `noemit`(a), suppresses the emission of progress events from a. Although a emits no events, it can still be preempted or canceled at its suppressed async points. This combinator is asynchronous and emits a *single* async point immediately after a completes execution, regardless of whether a was synchronous or asynchronous.

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 has an arbitrary number of async points. A time-limit for the *first* portion of the game can be encoded by the following.

$$\texttt{any}(\texttt{delay}(\mathit{limit}),\; \texttt{seq}(\texttt{noemit}(\mathit{game}_1),\; \mathit{game}_2))$$

Here, the `delay` arrow will register a handler for a timer event and immediately yield, at which point $game_1$ begins to execute. Any progress event emitted by $game_1$ is suppressed by `noemit`; however, an async point is emitted by the completion of $game_1$. If the timer completes, the sequence is canceled; if $game_1$ completes, the timer is canceled and $game_2$ begins to execute.

Unfortunately, such expressive semantics are not readily available using promises, as they do not support a natural way to emit progress events to prevent sibling promises from resolving, nor do they support cancellation (which is required for efficiency in most cases and correctness in others).

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`, denoted `fix`(ω `=>` 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 referencing a' from within the proxy arrow a. When execution hits the proxy arrow, it simply executes its reference with the proper arguments. This method allows a logically *infinite* arrow to be constructed in constant time and space.

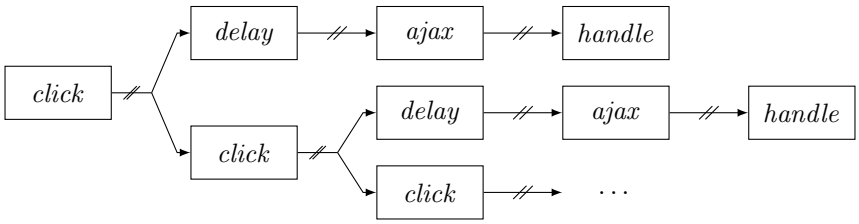

Figure 14: A recursive arrow composition (unwrapped recursion).

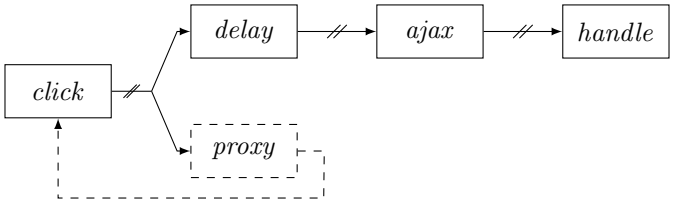

Figure 15: A recursive arrow composition (with a proxy arrow).

Figure 14 and Figure 15 show the arrow composition described in Section 2.1 Figure 5, which cannot be constructed without recursion.

Our previous work [5] included an additional combinator, `repeat`, for a restricted form of executing an a arrow *at least* once. 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 dataflow of the resulting arrow is demonstrated in Figure 16.

The combinator expects the result of a to be tagged union of the form $\langle loop : v',\ halt : v'' \rangle$, 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. This type is discussed further in Section 4.

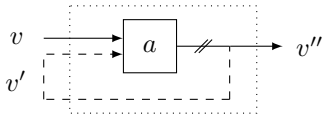

Figure 16: Dataflow diagram for the `repeat` combinator.

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 follows, where the arrow *repeatTail* takes the result of a (a tagged union) and either returns a value of type α or throws a value of type β. *repeatTail* is implemented as a lifted function, shown in Figure 17.

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

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

Figure 17: The implementation of the arrow *repeatTail*.

The `try` combinator will pass the value v' recursively to ω, or pass the thrown value v'' to the identity arrow. Notice that any invocation of a is **not** 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 in Figure 18. To demonstrate the use of the progress object p, we give the CPS encoding for the `delay` constructor, the `any` combinator, and the `try` combinator in Figure 19, Figure 20, and Figure 21, respectively. For brevity, we only show a binary version of the `any` combinator. In practice, this combinator can wrap $n \geq 1$ arrows.

```
call(x, p, k, h) {
    try {
        // Code for spreading arrays into arguments, type checking
        // of x and y would occur at this point, but has been omitted
        // for brevity.
        var y = f(x);
    } catch (e) {
        // Error continuation
        return h(e);
    }

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

Figure 18: Encoding for `lift`(f) - dynamic type checks omitted.

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

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

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

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

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

Figure 20: Encoding for `any`(a_1, a_2).

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. 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.

```
call(x, p, k, h) {
    // Create a new progress subtree which can be canceled independently.
    // Cancelling p should cancel branch; progress on branch advances p.
    const branch = new Progress();
    p.addCanceler(() => branch.cancel());
    branch.addObserver(() => p.advance());

    // Original error callback is called if success or error arrows yield
    // an error - this allows nesting of error callbacks.
    a.call(x, branch,
        y => as.call(y, p, k, h),
        z => {
            // "unwind" the stack by removing any event handlers registered
            // by execution of the protected arrow.
            branch.cancel();
            af.call(z, p, k, h);
        }
    );
}
```

Figure 21: Encoding for $\texttt{try}(a, a_s, a_f)$.

The `try` combinator creates a fresh progress object for the protected arrow. This progress object may be canceled independently from the parent progress object should the protected arrow yield an error, which prevents the cancellation of sibling arrows which lie outside of the `try` combinator.

4. Type Inference

In this section, we introduce the type system of our arrows library, which is carefully designed to ensure that it not only accepts the targeted applications of our arrows library but also permits efficient type inference. As a dynamic language, JavaScript does not have type annotations and the types of its expressions are checked at runtime. Inferring types for full JavaScript statically is extremely difficult due to features such as higher-order functions, subtyping, mutable states, objects, and prototypes. However, type inference for arrows is not only possible but also efficient for the following design choices.

- Arrow types are first-order in the sense that the input and output types of arrows do not contain function types.
- Constructed arrows are typed via user annotations or builtin types.
- The return value of each lifted function is dynamically checked to ensure that it is consistent with the declared return type. This allows us to avoid checking the body of the lifted function, which is plain JavaScript code.
- The type of a composed arrow is inferred through solving the constraints derived from the arrow, which is modular.

In Section 4.1, we define the types of values which can be consumed or produced by an arrow. In Section 4.2, present the typing rules for arrow constructors and arrow combinators The details of the type inference algorithm and its correctness are given in Section 5 and 6.

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.

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

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.

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

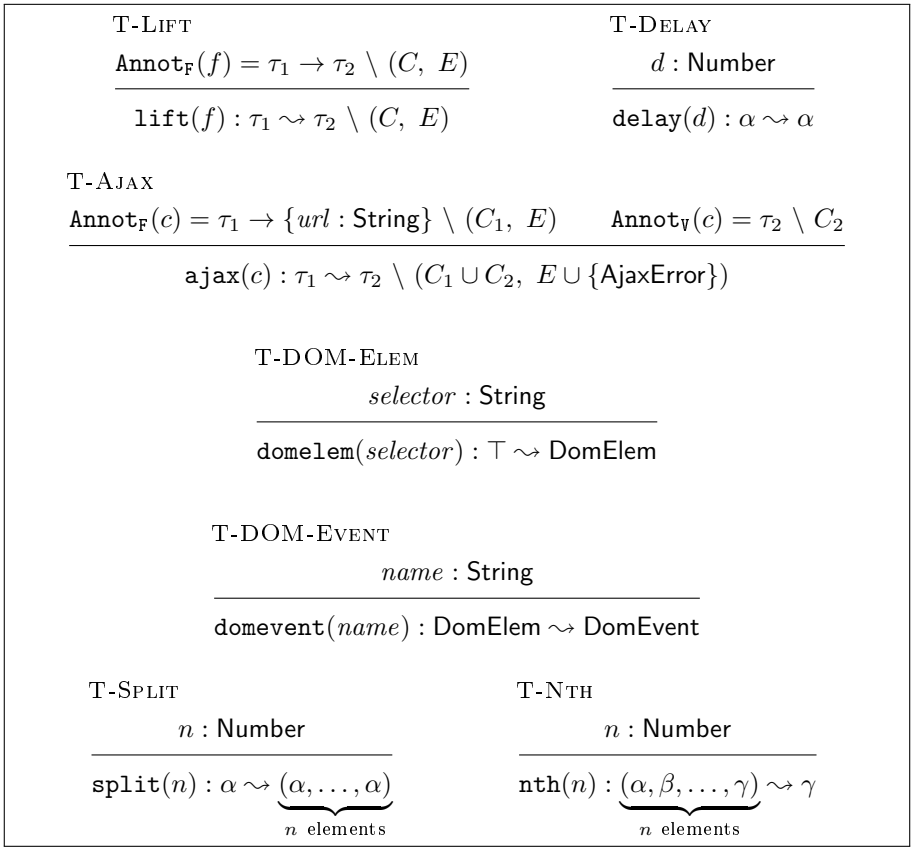

Figure 22: Typing rules for arrow constructors.

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 a significant value. The type $\top$ is also commonly 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 $\langle loop : \tau_1,\ halt : \tau_2 \rangle$ 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;

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 23: Typing rules for arrow combinators.

otherwise, a produces a final value v_2 of type τ_2.

An array type with homogeneously-typed 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

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

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 such as $\mathsf{String} \leq \mathsf{Number}$ or $\mathsf{Number} \leq (\mathsf{Number},\ \mathsf{Number})$.

The constrained arrow type is similar to the constrained type $\tau \setminus C$ introduced by Eifrig et al. [6], 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. Note that we use constrained types since more concrete solutions to the subtyping constraints may be too restrictive. For example, from the subtyping constraint $\alpha \leq \{x : \mathsf{Number}\}$ we can infer the type $\{x : \mathsf{Number}\}$ for α but that would exclude other solutions such as $\{x : \mathsf{Number},\ y : \mathsf{Bool}\}$.

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 22 and Figure 23, respectively. For brevity, the typing rules have the implicit assumption that if $a : \tau_{in} \rightsquigarrow \tau_{out} \setminus (C,\ E)$, 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 annotation 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.

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 constraint set $\{\alpha \leq \tau_1,\ \tau_2 \leq \beta\}$ ensures that an arrow of type $\tau_1 \rightsquigarrow \tau_2$ can be used whenever $\alpha \rightsquigarrow \beta$ is expected.

4.3. Consistency

In this section, we present the definition of constraint set consistency. An arrow is *consistent* if its constraint set is consistent (if no constraint in the constraint set produces an *immediate inconsistency*). For example, $\mathsf{String} \leq \mathsf{Number}$ 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 24. We refer to the closure of C as *closure*(C).

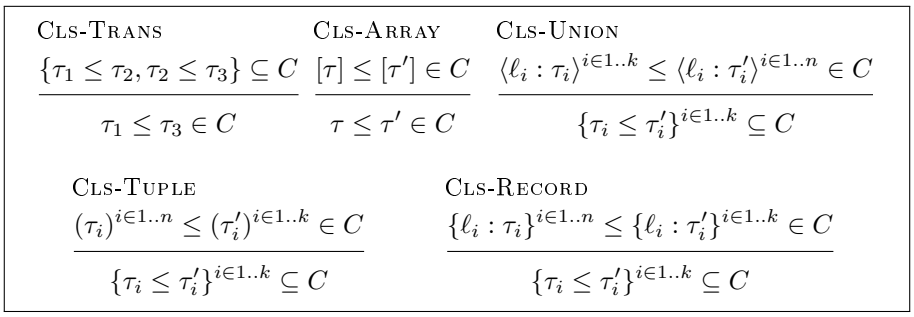

Figure 24: Constraint set closure rules, where $k \le n$.

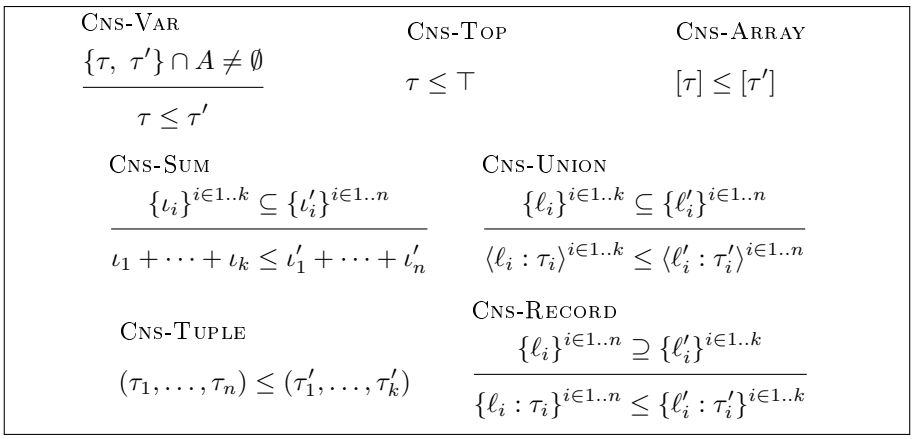

Figure 25: Constraint set consistency rules, where $k \le n$.

Definition 2 (Consistent)**.** A constraint set C is *consistent* if every constraint in *closure*(C) is consistent. A consistent constraint must match one of the forms given in Figure 25.

Rule (Cls-Trans) ensures that subtype constraints are transitive. For example, $C = \{\mathsf{String} \le \alpha,\ \alpha \le \mathsf{Number}\}$ contains only consistent constraints. However, $\mathsf{String} \le \mathsf{Number} \in \mathit{closure}(C)$ 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 value from a set of types T and an arrow consuming a value from a set of types T′ to be composed when $\mathrm{T} \subseteq \mathrm{T}'$.

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, union, tuple, and record depth subtyping is enabled by rules (Cls-Array), (Cls-Union), (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. For example, α has an impossible lower bound in $C = \{\alpha \leq \mathsf{String},\ \alpha \leq \mathsf{Number}\}$ as no lower bound of both String and Number exists. This case is handled by type simplification, discussed in Section 4.4.

4.4. 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 and inference overhead.

We simplify the type of an arrow immediately after its type is inferred inferred during composition (with one exception, discussed below). This prevents compositions of arrows from carrying constraints which are unreachable from the input type, output type, or set of exception types. Without simplification,

arrow types are noticeably larger and type inference is noticeably slower as closure calculation and consistency checks are at least linear (usually much larger) with the size of the arrow type.

As a motivating example, consider the following composition involving an arrow a of type $\mathsf{DomEvent} \leadsto \top$.

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

The resulting arrow takes as input a 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.

$$\begin{gathered}\alpha \leadsto \delta \setminus (\{\ \mathsf{DomEvent} \leq \mathsf{DomEvent},\ (\alpha,\ \alpha) \leq (\mathsf{DomElem},\ \beta),\\ (\top,\ \beta) \leq (\gamma,\ \delta),\ \alpha \leq \mathsf{DomElem},\ \alpha \leq \beta,\ \top \leq \gamma,\\ \beta \leq \delta,\ \alpha \leq \delta\ \},\ \emptyset)\end{gathered}$$

The first constraint, $\mathsf{DomEvent} \leq \mathsf{DomEvent}$, is introduced by the inner `seq`. The next two constraints $(\alpha,\ \alpha) \leq (\mathsf{DomElem},\ \beta)$ and $(\top,\ \beta) \leq (\gamma,\ \delta)$, are introduced through the `seq` combinator and the `split` and `nth` arrows. 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.

$$\alpha \leadsto \delta \setminus (\{\ \alpha \leq \delta,\ \alpha \leq \mathsf{DomElem}\ \},\ \emptyset)$$

A sample application implementing the game *Memory* (discussed in 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. Recursively composed arrows add four additional constraints on the input type α and the output type β to the arrow's constraint set *after* the type of the inner arrow is inferred. In this case, it

is incorrect to simplify the type of the inner arrow during composition, as it may remove a constraint involving α or β which will become necessary to the recursion, but appears locally unnecessary.

For example, consider the arrow with the recursive definition

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

where a has type $\iota_1 \rightsquigarrow \iota_2$. The typing of this arrow is shown below. Because $\{\iota_2 \leq \alpha,\ \alpha \leq \iota_1\} \in C$, $\{\iota_2 \leq \iota_1\} \in \mathit{closure}(C)$, 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 the constraint $\iota_2 \leq \alpha$ is simplified away and the type of the recursive arrow is incorrectly inferred to be $\alpha \rightsquigarrow \beta \setminus (\{\alpha \leq \iota_1\},\ \emptyset)$.

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 JavaScript's 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 define the operational semantics in Section 5.2. A translation from concrete arrows to abstract arrows is presented in Section 5.3.

5.1. Abstract Arrows

Figure 26 defines the complete abstract syntax. We extend the lambda calculus with sequencing, tuple values, tuple projection, tagged expression, and

e	$::=$	x	variable
	$\mid$	$e_1\ e_2 \mid v_f\ e$	application
	$\mid$	$e_1;\ e_2$	sequence
	$\mid$	$(e_1, \ldots, e_n) \mid e[j]$	tuple & projection
	$\mid$	$\ell(e) \mid \textbf{case}\ e\ \textbf{of}\ \{\ \overline{\ell(x) \Rightarrow e}\ \}$	tag & matching
	$\mid$	$\texttt{fix}(\lambda\omega.e_a)$	fixed-point combinator
	$\mid$	$e_a\ \bullet\ (e,\ e_p,\ e_k,\ e_h)$	arrow application
	$\mid$	$\texttt{async}\ v_e\ e_p\ e_k$	event registration
	$\mid$	$\texttt{adv}\ e_p$	event cancellation
	$\mid$	$\texttt{cancel}\ e_p$	
	$\mid$	$v \mid v_p \mid e_p \mid e_k \mid e_h \mid v_e$	
v	$::=$	$v_c \mid \lambda x.e$	constants, & abstractions
	$\mid$	$()$	unit value
	$\mid$	$\ell(v)$	tagged values
	$\mid$	$(v_1, \ldots, v_n) \mid \{\ell_1 : v_1, \ldots, \ell_n : v_n\}$	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
$\texttt{evn}$	$::=$	$\texttt{timeEv} \mid \texttt{ajaxEv}$	event types

Figure 26: Abstract syntax.

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 an abstract arrow is represented by an 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` and `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.

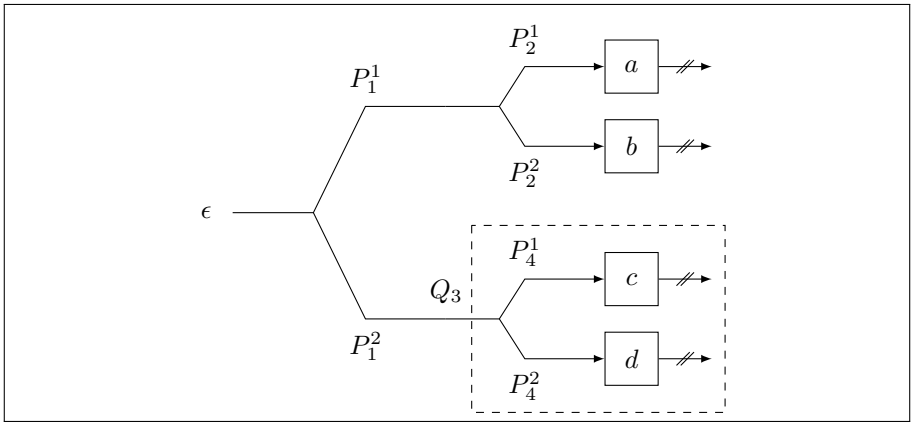

Figure 27: An example progress tree.

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 P_i^1 and 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 `try` combinator (using the progress token P_i^1) and the `noemit` combinator (using the progress token Q_i).

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 `try` and `noemit` combinators). Arrows may share a common prefix of abstract ar-

rows, which allows representing progress lists as a tree of progress tokens. The progress of the following arrow is illustrated in Figure 27.

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

Each branch of the `any` 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 de-registered.

The expression `adv` e_p 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. The expression `cancel` e_p cancels the arrows which are in the progress subtree e_p.

In reference to Figure 27, `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. `cancel` P_1^1 will cancel arrows a and b and `cancel` ϵ will cancel all four arrows. 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, denoted Δ, which maps event objects v_e to pairs of progress lists and continuations. A program starts with an empty event context.

$$\Delta ::= \emptyset \mid \Delta \cup \{v_e \mapsto (v_p,\ \lambda x.e)\}$$

In particular, if $\Delta \neq \emptyset$, $(\Delta,\ \langle v \rangle)$ may be be reducible while $(\Delta,\ v)$ cannot.

The evaluation context $\mathcal{E}$, described in Figure 28, 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
	$\mid$	`adv` $\mathcal{E}$	event expressions
	$\mid$	`cancel` $\mathcal{E}$	
	$\mid$	`async` $v_e\ v_p\ \mathcal{E}$	
	$\mid$	**case** $\mathcal{E}$ **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 28: Evaluation context.

Figures 29 and 30 define the operational semantics of abstract arrows. Control flow rules are given by rules (E-Seq) and (E-Case). Rule (E-Seq) encodes sequencing which evaluates left-to-right. Rule (E-Case) shows that case

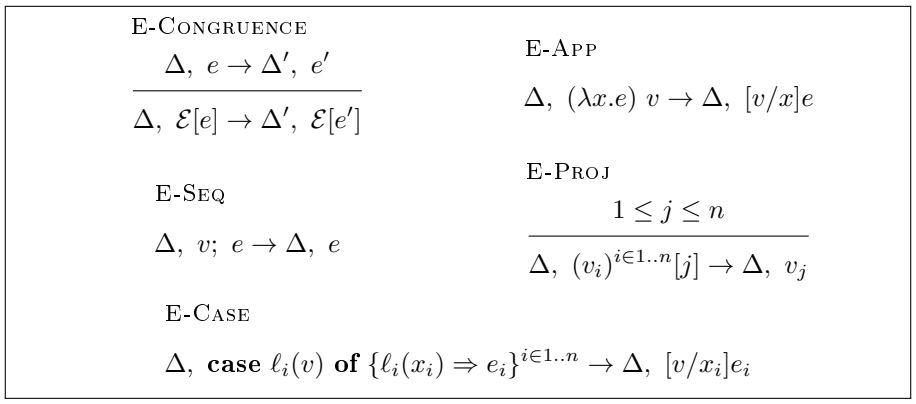

Figure 29: Operational semantics (lambda calculus and extensions).

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.

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 result of a host application must 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 runtime errors 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 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 consisting 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*

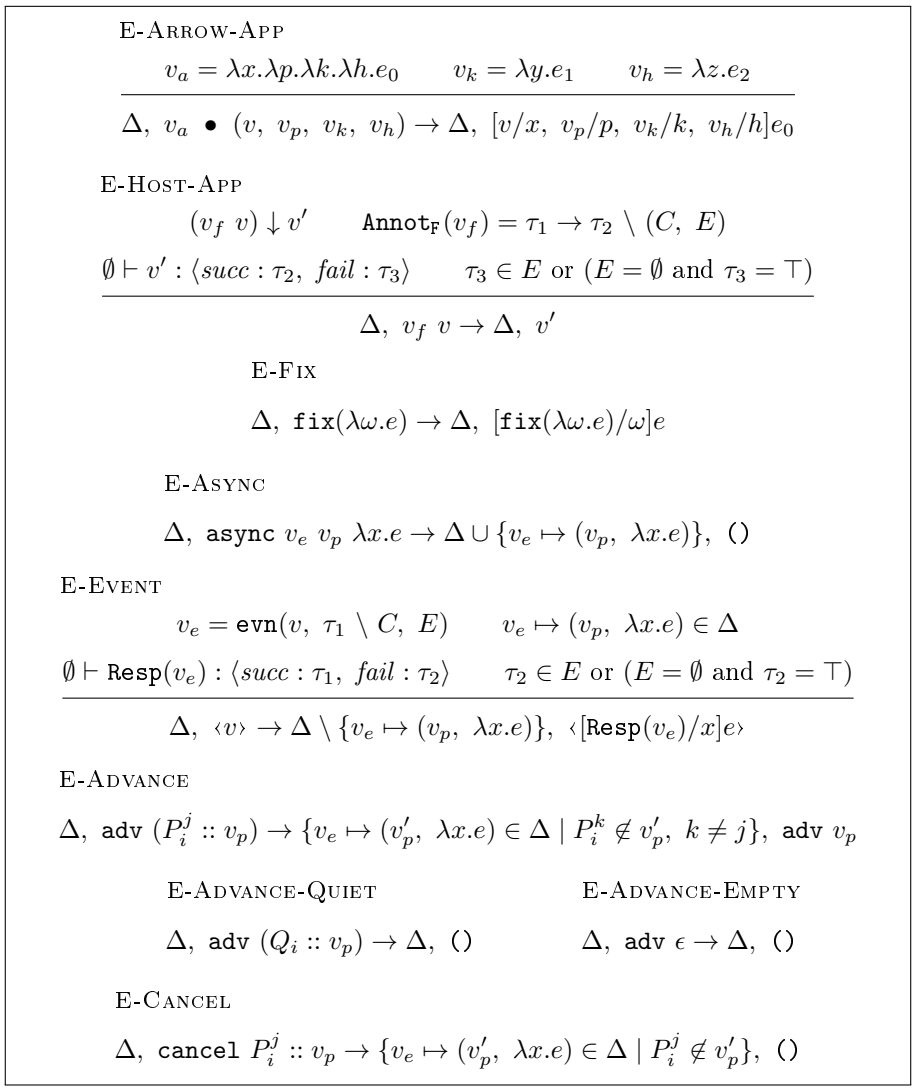

Figure 30: Operational semantics (host application and events).

(e.g. a timer elapsed or a network request 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 term is evaluated to completion before the rule can be applied again. It is assumed that v_e has completed when rule (E-Event) is applied - evaluation otherwise blocks. Similarly to host application, we insert a

runtime type check to ensure the value generated by the event conforms to the declared type of the event (which is necessary in practice when trusting data returned from a remote server during an Ajax event).

The rules (E-Advance) and (E-Advance-Empty) describe a specific form of event cancellation used by abstract arrows. The expression `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 progress token Q_i. 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.

Again, the arrows of Figure 27 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.

The rule (E-Cancel) describes a way to cancel the execution of a specific arrow. The expression `cancel` v_p will prune from the event context Δ all registered event handlers which are associated with a progress token that has a *suffix* of v_p (representing the ancestors in the progress tree). There is no need to cancel recursively, as in the advance rules, as the element at the head of the progress list v_p is sufficient to distinguish the remaining path. It is worth noting that the head of a canceled progress list will always be of the form P_i^1 as `cancel` occurs only within the translation of the `try` combinator.

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

Figure 31: Arrow translation rules (constructors).

Figures 31 and 32 define the translation from concrete arrows to abstract arrows. For simplicity, the translation rules for the n-ary combinators `seq`, `all` 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` and `all` combinators are fairly straightforward.

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

Figure 32: Combinator translation rules (combinators). Progress tokens P_i^1, P_i^2, and Q_i are fresh.

The combinator $\texttt{try}(a, a_s, a_f)$ executes the abstract arrow $[\![a]\!]$ with a unique progress list. This ensures that there is a subtree rooted at P_i^1 in the progress tree which can be canceled if the evaluation of the abstract arrow $[\![a]\!]$ yields an error. This essentially *unwinds* the stack of the protected arrow, removing any event handlers that it registered during its execution before its exceptional halt.

The combinator $\texttt{any}(a_1, a_2)$ executes the 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 $\texttt{noemit}(a)$ is translated to an abstract arrow $[\![a]\!]$ with a progress list prefixed by a unique progress token, which acts as a cancellation boundary for the executing

[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.

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 [7] for arrows translated to abstract syntax. Full proofs for each stated theorem appear in the appendix.

First, we establish that the translation of concrete arrows into abstract arrows preserves types (Theorem 1, stated below). For abstract syntax, we define an additional set of types composed of value types and functions over 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.

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.

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

Note that the type τ_3 can be a freshly generated type variable that serves as the 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.

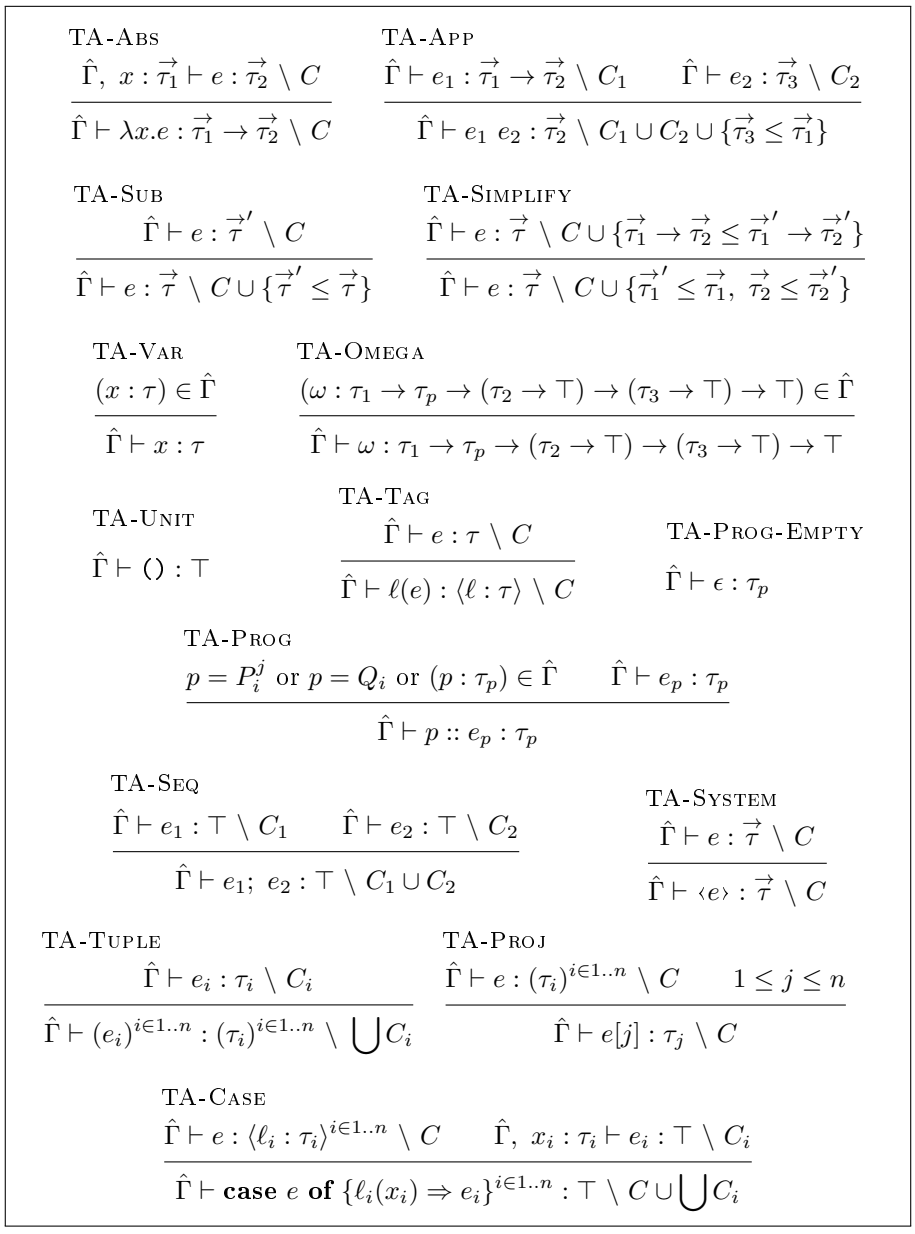

Figure 33: Typing rules for expressions in abstract syntax (lambda calculus and extensions).

The typing rules for arrows in abstract syntax are shown in Figure 33, Figure 34, and Figure 35. The typing rules have the implicit assumption that all type variables are fresh and if $\hat{\Gamma} \vdash e : \vec{\tau} \setminus C$, then C is consistent.

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

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

TA-Arrow-App

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

TA-Host-App

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

Figure 34: Typing rules for expressions in abstract syntax (arrow definitions and host application events).

We claim 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 and 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 $v_e \mapsto (v_p,\ \lambda x.e) \in \Delta$ the following properties hold

- $\emptyset \vdash v_e : \langle \mathit{succ} : \tau_1,\ \mathit{fail} : \tau_2 \rangle \setminus C$,
- $\emptyset \vdash \lambda x.e : \tau_3 \to \top \setminus C'$, and
- $C \cup C' \cup \{\langle \mathit{succ} : \tau_1,\ \mathit{fail} : \tau_2 \rangle \leq \tau_3\}$ 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

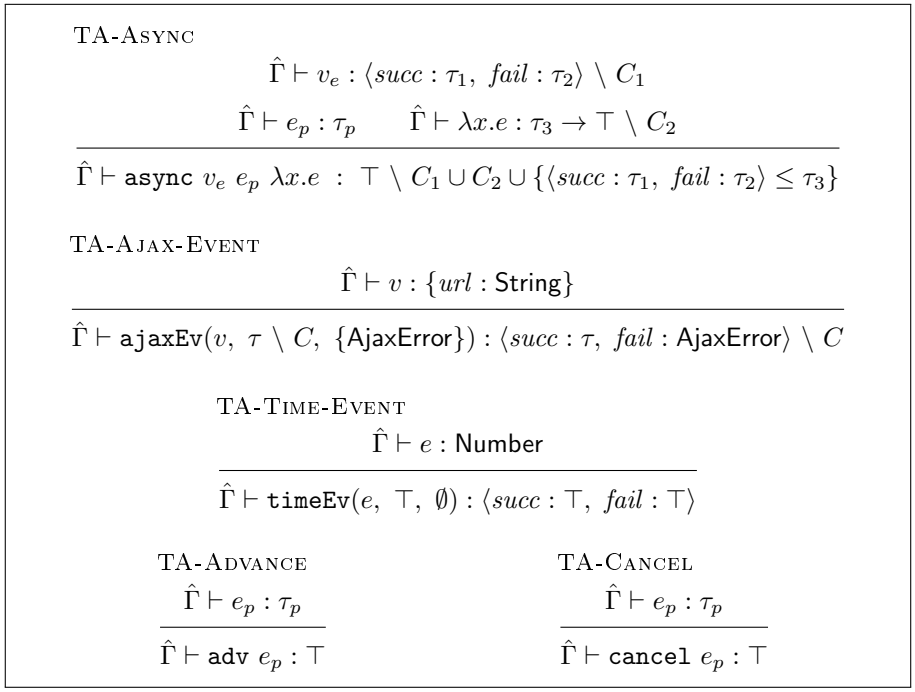

Figure 35: Typing rules for expressions in abstract syntax (events).

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.

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

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

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

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).

Theorem 5 (Soundness)**.** If $\emptyset \vdash a : \widetilde{\tau}$ and $\emptyset,\ [\![a]\!] \to^* \Delta,\ \hat{e}$, where $\to^*$ is a reflexive and transitive closure of $\to$, then one of the following conditions holds.

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

7.1. Implementation

Our arrows library consists of less than 1500 lines of code (900 of which implement the type checker), excluding the automatically generated type annotation parser. This library supports all versions of JavaScript. While our examples and library are written in ECMAScript 6 syntax, they have been automatically translated to syntax compatible to earlier versions of JavaScript. The primary platforms for our arrows library are browsers. However, it can also be used in platforms such as *Node.js* for server applications. For example, we have implemented an arrow constructor for database queries in Node.js and its implementation is very similar to that of the `ajax` arrow. In fact, this database arrow is used in the implementation of the backend for the *pagination* example listed on the project website.

7.2. Compatibility with existing code

Arrows are compatible with existing code implemented with callbacks and promises. Applications using arrows can call libraries implemented with callbacks or promises normally. In fact, our example applications use jQuery to access HTML elements. Program components implemented with arrows can be part of an application that also uses callbacks or promises. However, callbacks or promises cannot be used as arrows, which must be created through arrow constructors. In our experience, applications should use only arrows to encode the asynchronous logic in order to achieve more predicable semantics.

7.3. Sample Applications

In the following, we refer to a set of five non-trivial sample applications described below. The full source of each application is available at the project homepage[3].

Memory. A game played with a grid of sixteen cards in which the player must choose two cards until all matching pairs have been selected.

Whack-a-Mole. A game played with a grid of nine image elements. Each element is controlled independently by an arrow which displays a clickable mole for a randomly selected period of time. The game ends when either after the user has clicked on ten visible moles, or fifteen seconds have elapsed.

Music Player. A small music player that supports pausing and moving forwards and backwards through a fixed playlist.

Sorting Animation. An animation which randomly permutes (using the Fischer-Yates shuffling algorithm) then sorts (using Bubble Sort) a series of elements by width. This application is a semi-direct re-implementation of an Arrowlets application [4].

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

Pagination. An Ajax-driven application which displays a table of results from a remote server. The user can filter a set of results by keyword, and then page forward and backward through the filtered result set. This application includes the following three optimizations.

1. Cache responses from the remote server so that the same request does not require an extra network round-trip,
2. attempt to pre-fetch the next page of results while the application is idle on the current page, and
3. do not send an Ajax query until 400ms after the user stops typing in order to prevent spurious remote requests.

This application is described in detail by Fritz, Antony, and Zhao [8].

7.3.1. Application Complexity

Each application described above is implemented once using callbacks (for a baseline), and once using arrows. For comparison, the full source of each callback-only implementation is also available at the project homepage.

The number of lines of code (including blank lines and comments) for each application is given in Figure 36.

App	Shared	Arrows	Composition	Callbacks
Memory	91	38	38	54
Whack-a-Mole	31	80	60	74
Music Player	32	100	30	91
Sorting Animation	41	62	25	72
Pagination	0	122	42	72

Figure 36: The number of lines of code required to implement the sample applications.

The second column, **Shared**, gives the number of lines of code common between the implementation using arrows and the implementation using only

callbacks. The third column, **Arrows**, gives the total number of lines for the arrow implementation, including the definition of functions lifted into arrows. The fourth column, **Composition**, shows the number of lines spent on composing the arrow (excluding definitions of lifted functions and helper functions). The fifth column, **Callbacks**, gives the total number of lines for the callback-only implementation.

The implementations for Memory and the sorting visualizer require fewer total lines when implemented with arrows when compared to a callback-only implementation.

The implementations for Whack-a-Mole, the pagination application, and the music player require a greater number of lines when implemented with arrows when compared to a callback-only implementation, but only because the implementation defines named functions with type annotations. The music player, for example, defines a function `play` taking a single argument `song` with the body `song.trigger('play');` and type $\mathsf{DomElem} \rightsquigarrow \top$. This function is defined over five lines. The music player defines thirteen similar functions.

The callback-only implementations are not strictly equivalent to the arrow implementations of the same application. In particular, the pagination application does not cancel the pre-fetch remote request when the user begins a new query or pages backwards in the callback-only implementation, but does in the arrow implementation. This is trivial to encode with arrows but proved difficult to do correctly using only callbacks. In addition, the same application contains a race condition in the callback-only implementation – two Ajax requests can be in-flight at the same time in such a way that the earlier request comes back last and displays stale results. Because arrows implicitly cancel arrows which lose a race, this behavior is not present in the implementation using arrows. Similarly, the callback-only implementation of the music player contains a bug which is not present in the implementation using arrows – when moving backwards through tracks quickly, it is possible to move the application into a state where no song can play without refreshing the page. The source of this behavior is not obvious (and is currently unknown).

The sorting visualizer requires explicitly-managed recursion and contains deep nesting (shown in Figure 37) when implemented with callbacks. When implemented with arrows, the animation timing is a linear composition (e.g. `seq`(*indent*, `delay`(n), *swap*, `delay`(n), *dedent*, `delay`(n))).

```
function animateSwap(onComplete) {
    setTimeout(() => {
        indent();
        setTimeout(() => {
            swap();
            setTimeout(() => {
                dedent();
                setTimeout(onComplete, ANIMATION_DELAY);
            }, ANIMATION_DELAY);
        }, ANIMATION_DELAY);
    }, ANIMATION_DELAY);
}
```

Figure 37: A portion of the sorting visualization implemented using callbacks.

The Whack-a-mole game implemented using only callbacks uses a global variable as a cancellation token. When resuming after a timeout, the token is compared against a local *expected* value and will return immediately if these values differ. Without this cancellation, timers from previous games will still be running during the next round and can silently influence element click handlers, user score, element visibility, and the game-over timer. As arrows support implicit cancellation, this technique is unnecessary.

7.3.2. Annotation Burden

For each application described above, we determined the number of functions for which a user supplied an annotation. Figure 38 shows these results.

The second column, **Arrows**, gives the total number of arrows which were created for this application (including arrows created via combinators). The third column, **Annotations** gives the **total** number of arrows for which an annotation is attached. This includes built-in arrows as well as functions lifted by the user. This number is also not *unique*, so each arrow, such as the identity

App	Arrows	Annotations	User Annotations
Memory	132	35	8
Whack-a-Mole	314	85	5
Music Player	178	41	11
Sorting Animation	144	42	3
Pagination	105	26	9

Figure 38: Number of annotations parsed in each application.

arrow annotated with type $\alpha \rightsquigarrow \alpha$, may be counted multiple times. The fourth column, **User Annotations**, counts the number of *unique* annotations that the user supplied.

The last column gives a sense of the burden placed on the developer to support types. In all of the sample applications, the burden is very small – only a handful of comments must be placed inside functions lifted into arrows.

Furthermore, the annotations themselves are in the majority of cases extremely simple. Many lifted functions simply require a single parameter or pair of parameters with a concrete type. A handful of annotations are a bit more complex, but are not difficult to write. For instance, the Whack-a-Mole game creates an arrow which simply throws a value on the tenth whack in order to stop all nine moles from running. The type of this arrow is $\top \rightsquigarrow \top \setminus (\emptyset,\ \{\top\})$. It reads global state, does not read its parameter, has no return value, and only the *presence* of the exception value is meaningful. The pagination application contains an arrow which fetches a value from a global cache with the type $\alpha \rightsquigarrow \beta \setminus (\emptyset,\ \{\alpha\})$ which either returns the value in the cache or throws the key on cache miss. The type of this function is parametric so that it can be used in multiple contexts (and the cache is not bound to a concrete type of values). Lastly, the sorting application's main arrow has the following type.

$$(\mathsf{Number},\ \mathsf{Bool},\ \mathsf{Number}) \rightsquigarrow \langle \mathit{loop} : (\mathsf{Number},\ \mathsf{Bool},\ \mathsf{Number}),\ \mathit{halt} : \top \rangle$$

This arrow has three parameters: the index of the current element, whether or not an inversion was found on this pass, and how many elements have been

previously sorted. This arrow returns the arguments for the next pass when sorting should continue, and no meaningful value when sorting has completed.

7.3.3. Inference Overhead

For each application described above, we determined the overhead generated by inferring types of arrows at composition time. Figure 39 shows these results. This excludes type checks which occur at arrow *execution time*, which are discussed in Section 7.3.4.

App	Arrows	Disabled	Enabled	Parsing
Memory	132	9.28	42.06	4.64
Whack-a-Mole	314	13.58	100.98	8.42
Music Player	178	9.82	66.66	7.54
Sorting Animation	144	9.14	50.22	10.34
Pagination	105	10.02	53.42	8.52

Figure 39: The elapsed composition-time in each application (time in milliseconds).

The third column, **Disable** gives the time required to compose the arrow with type checking disabled. The fourth column, **Enabled** gives the time required to compose the arrow when inferring (and minimizing) types. The fifth column, **Parsing** shows how much of this time is required to read and parse type annotations in functions.

These measurements are the median of fifty runs in Google Chrome Version 54.0.2840.71 (64-bit). No warm-up runs were measured in order to ensure that the JIT cache stays cold between runs (as it would occur in practice).

These results show that the inference overhead is a function of both the number of *total* inferred types as well as the *size* of these types. The types of the first four applications are relatively simple. The median inference overhead *per arrow* is around $340\mu s$ for these applications. However, the pagination application has a median inference overhead per arrow of $500\mu s$. This is because the expected value returned by the server has the following (rather large) type

which must be unified multiple times during composition.

$$\{query : \mathsf{String},\ prev : \mathsf{Number},\ next : \mathsf{Number},\ count : \mathsf{Number},$$
$$rangeLeft : \mathsf{Number},\ rangeRight : \mathsf{Number},\ results : [\{$$
$$id : \mathsf{Number},\ name : \mathsf{String},\ category : \mathsf{String},\ subCategory : \mathsf{String},$$
$$pricePerUnit : \mathsf{Number},\ margin : \mathsf{Number}\}]\}$$

Although there is an increase in runtime when inferring arrow types, this cost is paid only once at application startup. We find this cost to be negligible in the context of asynchronous applications (especially in the browser) which spend the majority of their time blocking on external input (and, in the case of browsers, waiting at startup for remote dependencies to download).

In addition, this cost is generally paid only *during development*. If the startup cost is considered too high for a production environment, type checking can be disabled with no visible difference to the end user of an application. Types are used to generate more specific error messages when *composing* and *maintaining* arrows. These messages, however, will provide no benefit to the user of a broken application containing ill-composed arrows.

Currently, the implementation for type inference uses fixed-point iteration over a constraint set before pruning in order to attain the closure. This implementation is unoptimized, and these results are not necessarily indicative of the true overhead of the *technique* in general. Our future work includes an attempt to reduce this overhead.

7.3.4. Runtime Type Checking Overhead

For each application described above, we determined the overhead generated by the type checks inserted at the boundaries of lifted functions and Ajax arrows. Figure 40 shows these results.

The second column, **Total Time** gives the total sampled runtime of the arrow under execution. The third column, **Checks**, gives the number of runtime type checks performed. The fourth column, **Overhead**, is the total elapsed time spent performing runtime type checks.

App	Total Time	Checks	Overhead
Memory	38.065s	522	3.16ms
Whack-a-Mole	56.422s	752	4.16ms
Music Player	39.781s	1,065	8.35ms
Sorting Animation	48.034s	1,553	9.12ms
Pagination	26.540s	837	10.66ms

Figure 40: The total execution-time overhead in each application.

Each application was executed in a way indicative of its standard use. While the Memory arrow was running, the user selected twenty pairs of cards including three matches. While the Whack-a-mole arrow was running, the user won two games then lost two games. While the music player arrow was running, the user let a track play for ten seconds then skipped to the next track. While the sorting application arrow was running, the user shuffled the array three times, then sorted, then shuffled three more times, then sorted again. While the pagination arrow was running, the user entered a keyword with around one-thousand results, paged ten pages forward and five pages backwards, then repeated three times.

For each sample application, the time spent performing type checks during arrow execution is negligible, contributing below 0.05% of the total runtime. The median overhead *per type check* for the first four applications is $5.75\mu s$, and the median overhead per type check for the pagination application is $12.74\mu s$.

The number of type checks inserted for the music player is high as one arrow continually executes in order to update a progress element when a song is playing. The sorting animation also has a large number of type checks, as every step in the shuffle and sorting visualization requires multiple function calls (select a pair of elements, indent selected, change color of selected, perform swap, dedent selected, change color of selected).

7.4. Usability Study

In order to evaluate the usability of this library in practice, we asked a group of participants to add the same feature to two (behavior-identical) versions of a program – one written using only callbacks, and one written using arrows. The program given to the participants implemented an *image carousel* which loaded the previous or next image from a sequence (stored on a remote server) after a button click. They were asked to modify the program so that the carousel automatically advances on a timer. If the user presses the previous or next buttons, the timer should pause and allow the user to control the carousel explicitly. The timer should resume after the user is done interacting with the buttons.

From this group, fifteen participants submitted both versions. Figure 41 provides metrics on size and correctness of submissions organized by participant. The first group of columns gives the total number of lines of code (including blank lines and comments) for each submission. The cell containing the number of lines of code for the arrows solution is shaded if it was no larger than the callbacks solution. The second group of columns notes whether or not each submission included the requested feature such that it was *functional*. A feature with observable bugs is not considered functional.

While nine of the fifteen participants (60%) were able to modify the given arrows-based code as instructed, only seven of the fifteen participants (47%) were able to do so using only callbacks. Five of the fifteen participants (33%) were unable to correctly modify either template. Comparing the submissions from participants that completed both, the arrows solution was on average smaller.

The arrows-based submission from Participant #3 was nearly correct, but contained the incorrect expression

$$\mathit{loadImage}.\texttt{seq}(\mathit{next}.\texttt{after}(1000)).\texttt{any}(\mathit{blockOnClick})$$

instead of the working expression

$$\mathit{loadImage}.\texttt{seq}(\mathit{next}.\texttt{after}(1000).\texttt{any}(\mathit{blockOnClick}))$$

Participants	Lines of Code		Functional	
	Callbacks	Arrows	Callbacks	Arrows
1	40	45	no	no
2	55	36	no	no
3	44	37	yes	no
4	55	42	no	no
5	64	57	no	no
6	70	69	no	no
7	50	41	no	yes
8	49	42	no	yes
9	94	49	no	yes
10	39	39	yes	yes
11	74	43	yes	yes
12	68	52	yes	yes
13	39	56	yes	yes
14	64	60	yes	yes
15	64	84	yes	yes

Figure 41: The number of lines of code for each participant submission as well whether or not each submission contained the *functional* requested feature.

which works correctly – the idea seemed correct, but was off by one character due to expression nesting issues. This submission was considered non-functional.

The callback-based submissions from Participants #7, #8, and #9 contained bugs related to event cancellation. In some solutions, the click handlers were registered more than once, but no attempt was made to remove previously bound handlers. This made the current image skip forward in the sequence by larger and larger offsets. In other solutions, no attempt was made to make the timer and buttons mutually exclusive, resulting in obvious and user-observable asynchronous interference. The arrows-based submissions did not contain issues of this nature, as the arrows handle event de-registration implicitly.

Even with a small sample set, we believe that these results show that arrows can be leveraged to create asynchronous programs that are easier to correctly implement and correctly maintain.

8. Related Work

8.1. Arrows

Arrows [1] were first formalized as a generalization of monads [2]. 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.

Additional arrow combinators [9] have either direct or similar counterparts in our library. For example, `***` in corresponds to `all` and `&&&` corresponds to our `fanout` combinator, which is encoded by sequencing `split` with `all`. We do not have a choice combinator like |||, but our `try` combinator can achieve similar effect, as discussed in Section 3.2.

Alternate arrow calculi exist with only four constants satisfying five obvious laws [10, 11]. 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.

Arrows in Haskell. Arrows were initially implemented in Haskell using type classes similar to monads. However, the point-free style of arrows is awkward to use. A more convenient syntax for arrows was introduced by Paterson [12] as an extension to Haskell so that arrows can be defined using constructs similar to the **do** notation of Haskell. The main construct in this extension is arrow application `proc p -> e1 <- e2`, which roughly translates to `(p => e2).lift().seq(e1)` in our syntax (provided `p` and `e1` do not have the same free variables). It would

be useful to have similar syntax for JavaScript arrows, though a preprocessing step might be needed.

Arrowlets. Arrowlets is a JavaScript library for using arrows [4], 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.

Apart from our formalization of semantics and the addition of a type system, our execution semantics differ in the two following areas. First, we have altered the encoding of arrows to carry along an error continuation in addition to the happy-path continuation. This allows us to build the `try` combinator, which subsumes the semantics of ES6 Promises. Before this addition, Arrowlets would silently swallow errors just as vanilla callbacks do without the addition of explicit error-handling code spanning callbacks. Second, we have added a mechanism to change the semantics of concurrently executing arrows. We found that neither the semantics of the `any` combinator (first to progress) provided by Arrowlets nor the semantics of the `Promise.race` method (first to complete) encoded a sufficient set of programs by themselves. We required a way to encode both methods without the addition of an alternate `any` combinator. The addition of `noemit` provides a mechanism through which an individual arrow can re-define its definition of *progress* so that both ends of the spectrum (as well as all middle points) can be encoded.

Functional Reactive Programming (FRP). FRP [13] 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 [14] and Flapjax [15] 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 [16]. Arrows for FRP was further refined in a model called causal commutative arrows [17], which can be optimized through a transformation to causal commutative normal form with significant improvement in performance over conventional implementation of arrows.

8.2. Asynchronous 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 exist. Callbacks can be composed and registered to be invoked when the desired value is actually computed. Promises allow callbacks to be registered to multiple *states* (e.g. success, failure, done) of the promised value, so that different control flow paths can be taken depending on the outcome of the asynchronous operation.

While it is a successful pattern for asynchronous callbacks, it is less powerful than the composition of arrows which can create arbitrary control flow graphs through combinators.

ES6 Promises. Promises[4] 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.

[4] See `http://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects` for the formal specification.

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 an emphasis on the *values which they proxy*, where arrows place an 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 promise. Unlike the `any` combinator, `Promise.race` does not abort the execution of the remaining promises. We believe the semantics of the `any` combinator to be more useful in practice.

Promises, unlike arrows, allow functions sequenced within a promise chain to return a new promise instance. When this happens, the resolution of the chain depends on the resolution of the new instance. This technique is demonstrated in Figure 42. Here, the promise chain on line 8 will continue to resolve with the value `x * 2` immediately, and the promise chain on line 9 will pause for 500 milliseconds before resolving with the value `x * 2`.

First-class arrows, on the other hand, are not supported – arrow composition must be defined *in-full* before it is executed. This restriction promotes a more clearly defined separation of concerns. With promises, sequencing logic is mixed with the implicit logic required to resolve its value. With arrows, sequencing

```
1 function f1(x) { return x * 2; }
2 function f2(x) {
3     return new Promise((resolve, reject) => {
4         setTimeout(() => resolve(x * 2), 500);
5     });
6 }
7
8 p.then(f1).then(g);
9 p.then(f2).then(g);
```

Figure 42: An example promise chain, showing that a function can either a value (line 1) or another promise which resolves later (lines 2-6).

logic is explicit in the composition and does not muddle the logic of the lifted functions. The behavior of the promises in Figure 42 can be encoded in arrows as the following.

$$p.\texttt{then}(f_1).\texttt{then}(g) \equiv \texttt{seq}(p,\ f_1,\ g)$$
$$p.\texttt{then}(f_2).\texttt{then}(g) \equiv \texttt{seq}(p,\ \texttt{delay}(500),\ f_2,\ g)$$

Instead of returning an arrow a from f_2, we insert a into the composition where f_2 would be executed (f_2 is encoded as a delay arrow preceding f_1). This collapses the arrow composition into a single level. This pattern works in general as an arrow can be created without being immediately executed (unlike promises).

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.

Async/Await. The keywords async and await[5] have been proposed to restore some imperative features which were lost with the prolific use of callbacks. Any

[5]See `https://tc39.github.io/ecmascript-asyncawait/` for the feature proposal.

function which is annotated as *async* is free to use the *await* keyword on a promise, which will effective yield the current thread of control to the event loop, then resume once that promise resolves. Async functions always return a promise (and, if they are implemented as returning a naked value, it is implicitly wrapped in a promise). An *await* expression either yields the promise's resolved value or raises the promise's rejection value.

```
function wait(t) {
    return new Promise((resolve, reject) => setTimeout(() => {
        if (t < 500) { resolve(t * 2); }
        else         { reject(t); }
    }, t));
}

async function test() {
    try {
        await wait(250) + await wait(750);
    } catch (err) {
        return err;
    }
}
```

Figure 43: An example use of async and await in JavaScript.

Figure 43 shows an example of an async function – calling `test` will yield a promise that resolves with the value 750 after one second. Each use of *await* blocks until the promise resolves – concurrent execution can still be achieved with explicit use of `Promise.all`.

Async and await features are merely sugar over the existing capabilities of promises – the await keyword sequences the remainder of the function as a resolution handler of the promise on which it awaits. Because these features provide no additional semantics, they are equivalently as limited as promises.

Factors. Factors [18] 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 immedi-

ately, 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. [19] present a small-step operational semantics strictly conforming to the ECMAScript (ECMA262-3) specification. 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 [20].

8.3. Static Analysis of Dynamic Languages

Our arrows library introduces an optional type system which helps users detect errors early in the development cycle. As dynamic languages such as JavaScript and Ruby have gained popularity in recent years, a number of systems have been develop to analyze programs written in these languages to achieve the same goal.

For instance, TeJaS [21] is a type system framework that is designed to retrofit customized type systems to JavaScript programs. This framework has been applied to analyzing the safety of third-party scripts [22], violation of private browsing in Firefox extensions [23], and the correctness of jQuery programs [24]. To handle features of the entire JavaScript, TeJaS has included complex type constructs such as recursive types, bounded quantification, union types, intersection types, special treatment of fields, and kinding. To type check a program, users may need to slightly refactor the program and add type annotations to functions, objects, and variables. TeJaS supports limited local type inference and provides syntactic sugar to lower annotation overhead.

Since TeJaS is a type system for the entirety of JavaScript, it is much more complex than ours, which does not have to consider problematic features such as first class functions, object prototypes, mutable fields, or object extension. Our arrows library checks the arguments and the result of a lifted function but it does not check the function body, which can be arbitrarily complex. This design choice is ideal for arrow combinators as users can partition the functionalities of an application into small functions with concentrated tasks while the arrows library can ensure that correct values are transferred between functions. Despite the simplicity of our system, we have not found it to be limiting in its intended applications. Also, our type system uses constrained types to encode recursively bounded quantification and it infers types for composed arrows, which very easily outnumber the lifted functions.

Other than type systems, errors in dynamic languages can be detected through program analysis. A recent work on static analysis of event callbacks in Node.js detects errors such as dead listeners, incorrect sequencing of synchronous and asynchronous calls, and incorrect sequencing of callbacks [25]. Their tool, RADAR, constructs an event call graph and performs an event-sensitive and listener-sensitive dataflow analysis. The tool seems to be able to detect known errors and has manageable false positives. While our arrows library does not perform a similar analysis to detect dead listeners or incorrect sequencing, the event semantics of both our frameworks are similar.

Also, the kind of errors detected by RADAR are less likely to happen if the programs are implemented with arrows. For example, dead listener error is caused by registering callbacks on event source that does not emit such event. Using arrows, the event source would be implemented as an element arrow and to listen to an event on that source, users need to sequence an event arrow and a handler arrow after the element arrow. If users attempt to sequence an event arrow after an incompatible event source, a typing error will be detected since the type of the event arrow specifies the type of element that can emit that event.

Incorrect sequencing of callbacks is less likely with arrows as well, since

arrow composition requires explicit scheduling of the arrows through `seq`, `any`, or `all` combinators. Users are less likely to have incorrect assumption that an asynchronous call will complete before the next synchronous or asynchronous call since they have to explicitly specify whether the ordering is sequential or concurrent, which is enforced by the arrow runtime.

Lastly, our separation of arrow composition and execution phases has some similarity to a style of static analysis called just-in-time static type checking for dynamic languages. Ren and Foster recently developed a static type checking tool called Hummingbird, which type checks Ruby code in the presence of metaprogramming [26]. The idea of Hummingbird is to utilize the metaprogramming code of Ruby to dynamically generate types, which are used to check each method before it is called. The computed types are cached until they are invalidated by changes in the type environment.

Unlike arrows, Ruby programs may not have distinct metaprogramming phase and actual computation phase. Therefore, a Ruby method may be type checked multiple times during its execution. Hummingbird also injects dynamic type checks to verify arguments passed to methods if the arguments come from unchecked code. While caching type information reduces overhead of Hummingbird, running a program with type check enabled still slows down a Ruby program up to five times.

Our arrows library, however, only needs to infer types once before actual asynchronous computation starts. The dynamic checks of arrows take a very small fraction of the overall runtime. Moreover, since the type of an arrow does not depend on runtime values, we can turn off type inference for deployed applications while Hummingbird may not have such an option since the types can be dynamically modified.

9. Conclusion

We have presented an arrows library for implementing asynchronous computation in JavaScript. Applications developed with arrows are more modular and

reusable. The program semantics resembles that of the sequential programs and the cancellation semantics of arrows results in more predictable program behavior and reduces the chances of producing undesirable side effects. Our arrows library is also expressive as it encodes most of the semantics of ES6 Promises including its error handling mechanism.

More importantly, our arrows library has a composition-time type checker which enables type-directed development. This optional type checker helps identify errors that are difficult to track in asynchronous JavaScript programs. The type checker can be disabled before application deployment without changing program semantics to eliminate the runtime overhead associated with types.

While our arrows library has demonstrated utility in a number of small applications, as future work, we plan to evaluate it in the context of larger applications and libraries. We believe that our arrows library provides a solid core and additional constructors and combinators can be developed incrementally to provide additional domain-specific features.

References

[1] J. Hughes, Generalising Monads to Arrows, Science of Computer Programming 37 (1998) 67–111.

[2] 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.

[3] S. D. Swierstra, L. Duponcheel, Deterministic, error-correcting combinator parsers, in: International School on Advanced Functional Programming, Springer, 1996, pp. 184–207.

[4] 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.

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

[6] 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.

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

[8] E. Fritz, J. Antony, T. Zhao, Arrows in commercial web applications, HotWeb 2016 (2016).

[9] J. Hughes, Programming with Arrows, in: 5th International Summer School on Advanced Functional Programming(LNCS 3622), Springer, 2005, pp. 73–129.

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

[11] 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.

[12] R. Paterson, A new notation for Arrows, in: Internationl Conference on Functional Programming (ICFP), 2001, pp. 229–240.

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

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

[15] 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.

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

[17] H. Liu, E. Cheng, P. Hudak, Causal commutative Arrows and their optimization, in: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP), 2009.

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

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

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

[21] B. S. Lerner, J. G. Politz, A. Guha, S. Krishnamurthi, TeJaS: retrofitting type systems for JavaScript, in: ACM SIGPLAN Notices, Vol. 49, ACM, 2013.

[22] J. G. Politz, S. A. Eliopoulos, A. Guha, S. Krishnamurthi, ADsafety: type-based verification of JavaScript sandboxing, in: USENIX Security Symposium, 2011.

[23] B. S. Lerner, L. Elberty, J. Li, S. Krishnamurthi, Combining form and function: Static types for JQuery programs, in: European Conference on Object-Oriented Programming (ECOOP), 2013.

[24] B. S. Lerner, L. Elberty, N. Poole, S. Krishnamurthi, Verifying web browser extensions' compliance with private browsing mode, in: European Symposium on Research in Computer Security (ESORICS), 2013.

[25] M. Madsen, F. Tip, O. Lhotak, Static analysis of event-driven node.js JavaScript applications, in: AMC SIGPLAN Notices, Vol. 50, ACM, 2015.

[26] B. M. Ren, J. S. Foster, Just-in-time static type checking for dynamic languages, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2016.

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 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.

$$b_{\downarrow}(\alpha) = \{\tau_i \mid \tau_i \leq \alpha \in C \text{ and } \tau_i \text{ is concrete}\}$$
$$b_{\uparrow}(\alpha) = \{\tau_i \mid \alpha \leq \tau_i \in C \text{ and } \tau_i \text{ is concrete}\}$$

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 and the upper bounds of a type variable β by applying the following transformations to C

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

where $(\vee\ \mathrm{T})$ 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.

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

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 $\tau \leq \tau'$, 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 α occurs in either 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. 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 variables 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 (normal or

exceptional). 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.

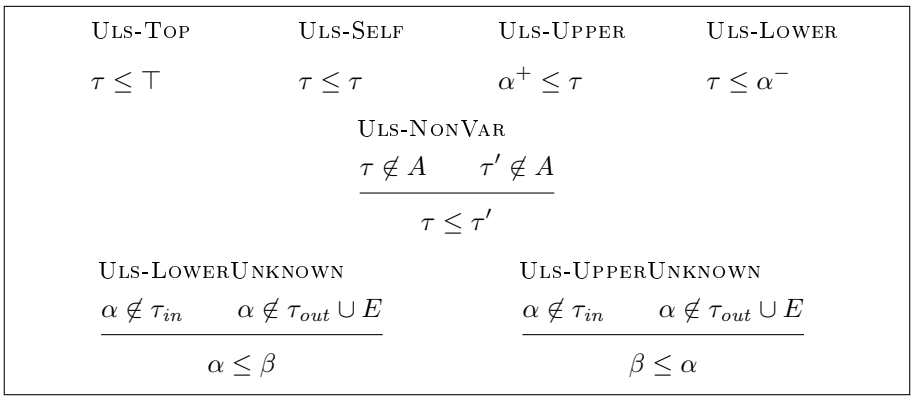

Figure A.44: Useless constraint elimination rules.

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

Appendix B. Proof of Theorem 1 (See page 50)

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,

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

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

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

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 $\{\mathit{url} : \mathsf{String}\}$, $C = C_1 \cup C_2$, and $E = E_1 \cup \{\mathsf{AjaxError}\}$. 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).

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

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.

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

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).

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

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.

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

The missing constraints $\{\tau \leq \tau_3 \mid \tau \in E\}$ 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 $C_1 \cup \{\tau \leq \tau_3^1 \mid \tau \in E_1\} \subseteq \hat{C}_1$ 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), (TA-Proj), (TA-Abs), (TA-App), and (TA-Tuple).

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

The additional constraints originate from application of rules (TA-Sub) and (TA-Simplify) which are used to make the type of 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

$$C = C_1 \cup C_2 \cup C_3 \cup \{\tau_2^1 \leq \tau_1^2,\ \tau_2^2 \leq \tau_2\ \tau_2^3 \leq \tau_2\} \cup \{\tau \leq \tau_1^3 \mid \tau \in E_1\}$$

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), (TA-Abs), (TA-Seq), and (TA-Cancel).

$$\begin{aligned}\Gamma' \vdash\ & [\![a_1]\!] \bullet (x,\ P_i^1 :: p,\\ & \lambda y.[\![a_2]\!] \bullet (y,\ p,\ k,\ h),\\ & \lambda y.\texttt{cancel}\ P_i^1 :: p;\ [\![a_3]\!] \bullet (y,\ p,\ k,\ h)) : \top \setminus \hat{C}\end{aligned}$$

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 $[a = \texttt{any}(a_1, a_2)]$. 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 51)

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 $[\langle v \rangle]$, $[\texttt{adv}\ v_p]$ *and* $[\texttt{cancel}\ v_p]$. For each case, the rules (E-Event), (E-Advance), (E-Advance-Empty), and (E-Cancel) guarantee $\Delta' \subseteq \Delta$. 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 51)

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), $\hat{e}' = [v/x]e_0$. We have the following by rules (TA-App) and (TA-Abs).

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

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 $(v_f\ v) \downarrow v'$. We have the following by rule (TA-Host-App).

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

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), $e' = \texttt{()}$. 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 $\hat{\Gamma} \vdash \texttt{()} : \top$ 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), $\hat{e}' = \texttt{adv}\ v_p$. We have the following by rules (TA-Advance) and (TA-Prog).

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

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.

Case [$\texttt{cancel}\ v_p$]. By rule (E-Cancel), $\hat{e}' = \texttt{()}$. By rules (TA-Cancel) and (TA-Prog), we have the following.

$$\frac{\hat{\Gamma} \vdash v_p : \tau_p}{\hat{\Gamma} \vdash \texttt{cancel}\ v_p : \top}$$

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 52)

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

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, $v_1 = \lambda x.e_0$. $\hat{e}' = [v_2/x]e_0$ 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

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

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

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

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 $\emptyset \vdash v' : \langle \mathit{succ} : \tau_2 \rangle$ or $\emptyset \vdash v' : \langle \mathit{fail} : \tau_3 \rangle$ 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), $\hat{e}' = [\texttt{fix}(\lambda\omega.e)/\omega]e_a$.

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

Case $[\langle\!\langle v \rangle\!\rangle]$. 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 $\emptyset \vdash \texttt{Resp}(v_e) : \langle \mathit{succ} : \tau_2 \rangle$ or $\emptyset \vdash \texttt{Resp}(v_3) : \langle \mathit{fail} : \tau_3 \rangle$ 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).

Case $[\texttt{cancel}\ v_p]$. By rule (E-Cancel), $\hat{e}' = \texttt{()}$ and $\Delta' \subseteq \Delta$.

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