Contracts for Async Patterns in JavaScript

Tim Disney

Cormac Flanagan

Abstract

Behavioral contracts are widely used in programming languages to specify and enforce the dynamic behavior of programs. In this paper we present an extension of contracts.js, a behavioral contract library for JavaScript, that enables programmers to specify temporal properties of programs. In particular, we describe *async* contracts that enforce when a function may be invoked with respect to the event loop.

1. Introduction

Behavioral contracts are widely used in programming languages including Eiffel [1], Scheme/Racket [2], and JavaScript [3–6] to specify and enforce the dynamic behavior of programs. Much of the work done recently in contract systems has been in extending the expressive power of contracts, for example to handle polymorphic specifications [7] or integrate with types [8].

In prior work, we proposed a general contract framework for specifying and enforcing higher-order temporal properties [9]. Here, we present several contracts that address specific temporal patterns commonly found in JavaScript programs.

Core to JavaScript's notion of temporality is the event loop. Unlike a preemptive multithreading language such as Java where the scheduler can switch control between threads at any point, programs in JavaScript process each event one after another. Each event is guaranteed to run to completion before returning control to the event loop, which then processes the next event in a queue. While the run-to-completion semantics of JavaScript is easier to reason about than threads, there is still plenty of room for surprising temporal bugs to bite.

One area temporal bugs can arise is when confusing *synchronous* and *asynchronous* functions. A synchronous function is a function that is called before returning control to the event loop whereas an asynchronous function is called on some later turn of the event loop.

As an example of a temporal bug that confuses synchronous and asynchronous functions consider the following API for a node.js program that provides a caching layer in front of file access (adapted from an example in Effective JavaScript [10]):

```
var readFile = require("fs").readFile;
var cache = new Map();

function readCaching(fileName, onsuccess) {
    if (cache.has(fileName)) {
        onsuccess(cache.get(fileName));
    }

    readFile(fileName, "utf8", function(err, data) {
        cache.set(fileName, data);
        onsuccess(data);
    });
}
```

As its name suggests, the node.js function `readFile` reads a file and asynchronously invokes its callback on some later turn of the event loop (once the file has been read from the disk). At first glance `readCaching` seems fine, it calls the `onsuccess` callback on a cache hit otherwise it first calls `readFile` before invoking `onsuccess` once the file reading operation completes.

The problem here is that `readCaching` implements an inconsistent API; sometimes the `onsuccess` handler is called asynchronously (when there is a cache miss) and sometimes synchronously (when there is a cache hit). Client code that is unaware of this inconsistency and expects the `onsuccess` to always be called asynchronously can have its assumptions violated leading to subtle bugs. Consider:

```
var obj = {};
readCaching("foo.txt", function(data) {
    obj.totalLength += data.length;
});
obj.totalLength = 0;
```

If `"foo.txt"` is not in the cache then this snippet works fine since the client has a chance to initialize `obj` before the handler is called. If, on the other hand, `"foo.txt"` is actually in the cache then the handler is called before the client code has a chance to finish initializing `obj`, which means that the final value of `totalLength` will be `NaN` (since in JavaScript `undefined + data.length` will be evaluate to `NaN`). Since the bug depends on what is in the cache, we have a source of nondeterminism that makes reproducing the failure difficult.

2. Async Contracts

To address this problematic temporal behavior we add *async* contracts to contracts.js, a higher-order JavaScript contract library. Contracts.js uses sweet.js [11], a macro system for JavaScript, to provide expressive syntax support around a

runtime contract library. We can then rewrite our problematic example by wrapping the function `readCaching` in the contract `(Str, (Str) --> ()) -> ()`.

This contract says `readCaching` is wrapped in a function contract (written `->`) that takes two arguments, a string (`Str`) and an async contract (`(Str) --> ()`) that takes a string and returns undefined. A function wrapped in the standard `->` contract can be invoked either synchronously or asynchronous whereas a function wrapped in the `-->` contract *must* be invoked asynchronously (ie. on some later turn of the event loop). Since `readCaching` does not obey this specification, when `onsuccess` is synchronously invoked on a cache hit the contract will throw an error blaming `readCaching` for violating its contract.

To implement async contracts we need a way to know on what turn of the event loop an event is currently executing. A simple way to accomplish this is to make a unique identifier for each event in the loop available to an async contract. Then the process of checking for async/sync behavior can proceed as follows:

1. Wrap the async function in its contract
2. Record the event loop id in which the wrapping took place
3. When the wrapped async function is invoked:
   - If the current loop id is equal to recorded loop id then raise blame
   - Otherwise continue execution

An example implementation of async contracts for just asynchronous checking (ignoring the domain and range contracts for simplicity) would look something like this:

```
function async(f) {
    var loopId = getLoopId();
    return function() {
        if (getLoopId() === loopId) {
            throw new Blame("Called synchronously");
        }
        // invoke the function normally
        return f.apply(this, arguments);
    };
}
```

While the function `getLoopId()` does not exist in JavaScript most JavaScript environments provide the means for us to implement `getLoopId()` ourselves. In particular node.js provides the function `process.nextTick(cb)` that invokes its callback before the next turn of the event loop. This allows us to implement `getLoopId()` directly; each time `getLoopId` is called the current loop id is returned and `process.nextTick` is used to queue up a callback that increments `loopId` before the next turn of the event loop occurs:

```
var loopId = 0;
function incLoopId() { loopId++; }
function getLoopId() {
    process.nextTick(incLoopId);
    return loopId;
}
```

In browser environments `nextTick` is not available but the `setImmediate` function could be used to a similar effect however it is only available in certain browsers and its standardization is contested. In any event, polyfills for `setImmediate` exist[1] that take advantage of clever tricks using `postMessage` (a function meant for cross-document messaging) and web workers.

Unsurprisingly, it is straightforward to implement the dual of an async contract: a sync contract that specifies the function must be invoked on the same turn of the event loop. The only change required is that the loop id when the function is invoked must be the same as when the function was wrapped in the `sync` contract. It is also straightforward to implement a contract that checks that its argument is consistently used either synchronously or asynchronously by recording how it was used the first time and then consistently enforcing the same behavior.

We have implemented async contracts in contracts.js[2]. Preliminary benchmarking suggests that asynchronous contracts impose minimal performance costs over traditional contract checking. More investigation is needed to determine the utility of asynchronous contracts but their ability to catch temporal violations as described in this paper is promising.

References

[1] Bertrand Meyer. Eiffel: The Language. *Prentice-Hall 1991*, 1991.

[2] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59. ACM, 2002.

[3] Matthias Keil and Peter Thiemann. Efficient Dynamic Access Analysis Using JavaScript Proxies. *arXiv.org*, pages 49–60, December 2013.

[4] Peter Thiemann and Matthias Keil. TreatJS: Higher-Order Contracts for JavaScript. URL `http://proglang.informatik.uni-freiburg.de/treatjs/`.

[5] Guillaume Marceau. rho-contracts. URL `https://github.com/sefaira/rho-contracts.js`.

[6] Tim Disney. Contracts.js. URL `http://contractsjs.org`.

[7] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Proceedings of the 2007 symposium on Dynamic languages - DLS '07*, pages 29–40, New York, New York, USA, October 2007. ACM.

[8] J G Siek and W Taha. Gradual typing for functional languages. *Scheme and Functional Programming*, pages 81–92, 2006.

[9] Tim Disney, Cormac Flanagan, and Jay McCarthy. Temporal higher-order contracts. In *ICFP '11: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, pages 176–188, New York, New York, USA, September 2011. ACM Request Permissions.

[10] David Herman. *Effective JavaScript*. 68 Specific Ways to Harness the Power of JavaScript. Addison-Wesley, November 2012.

[11] Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. Sweeten your JavaScript: hygienic macros for ES5. In *DLS '14: Proceedings of the 10th ACM Symposium on Dynamic languages*, pages 35–44, New York, New York, USA, October 2014. ACM Request Permissions.

[1] https://github.com/YuzuJS/setImmediate

[2] available at `http://contractsjs.org/`