Contracts for Async Patterns in JavaScript

Tim Disney

Cormac Flanagan

Abstract

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 specific contracts that address common temporal patterns found in JavaScript programs.

Core to JavaScript's notion of temporality is the event loop. JavaScript has no preemptive multithreading and all events run to completion. 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 example of a temporal bug (as described in Effective JavaScript [10]) is an API that confuses synchronous from asynchronous calls. For example, consider the following API for a node.js program that provides a caching layer in front of file access:

```
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);
    });
}
```

At first glance this function seems fine, it calls the `onsuccess` callback on a cache hit otherwise it first calls the underlying platform's asynchronous `readFile` function before invoking `onsuccess`. The subtle problem here is that sometimes `onsuccess` is called on a following turn of the event loop (when the cache is empty) and sometimes on the same event (when there is a cache hit). This means that code expecting `readCaching` to be asynchronous may have inconsistent state. Consider:

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

If none of the files are in the cache this works just fine but if there are any cache hits the shared object will not have completed initializing before the `readCaching` callback is invoked and `obj.totalLength` will be undefined. The user of `readCaching` is expecting the callbacks to be invoked on a subsequent turn of the event loop.

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 and thus allows us to rewrite our problematic example as:

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

@ (Str, (Str) ~> ()) -> ()
function readCaching(fileName, onsuccess) {
    if (cache.has(fileName)) {
        onsuccess(cache.get(fileName));
    }

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

The `@` wraps `readCaching` in a function contract (written `->`) that takes two arguments, a string (`Str`) and an async contract (`(Str) ~> ()`) that takes a string and returns undefined. The key behavior that an async function contract enforces is that the function must *not* be invoked on the current turn of the event loop. Since `readCaching` does not obey this specification on a cache hit that synchronously invokes `onsuccess` the contract with throw an error blaming `readCaching`.

To implement this async contract we need a way to reify the event loop. A simple way to represent the event loop is to have a unique identifier for each loop that an async contract can inspect. Then the process of checking for async/sync behavior can proceed as follows:

1. A function with an async parameter is called
2. Wrap the async parameter in its contract
3. Record the event loop id in which the wrapping took place
4. When the wrapped async parameter is invoked:
   - if the current loop id is equal to recorded loop id then raise blame
   - otherwise continue execution

An example implementation of this for just asynchronous checking (ignoring the domain and range contracts) 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 where the function must be invoked on the same turn of the event loop:

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

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` contracts for it to pass. It is also straightforward to implement a contract that checks that its argument is consistently used either synchronously or asynchronously by checking how it was used the first time and then consistently enforcing the same behavior.

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**.

[5] sefaira/rho-contracts.js.

[6] Tim Disney. Contracts.js.

[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