JavaScript verification with JaVerT: more examples

Anonymous authors

June 12, 2017

1 Overview

We have further examples in addition to the priority queue example of the paper. One of them is a pedagogical counter example on systematically, abstractly handling function closures and is presented in detail in Section 2. Another is a simple key-value map, shown in Section 3, where the main point is dynamic property access. To illustrate set reasoning, we verify operations on binary search trees, whereas for list reasoning, we verify an insertion sort algorithm.[1] The table below contains the statistics for these examples.

Example	**JS lines**	**JSIL lines**	**Verified specs**	**Time(s)**
Priority queue	40	912	10	9.2
Counter	19	412	4	3.9
Key-value map	21	470	22	4.7
BST	70	1033	6	8.5
Insertion sort	24	418	3	3.0
JS Internal functions	N/A	1048	186	5.1

Table 1: Verification statistics for the examples

These examples and all JavaScript programs in general heavily rely on the JavaScript internal functions. We list in the table also the breakdown for all of the internal functions that we have implemented in JSIL, axiomatically specified in JSIL Logic, and verified using JSIL Verify.

[1] These two examples we have in common with KJS, so we can compare. On a machine with an Intel Core i7-4960X CPU 3.60GHz and DDR3 RAM 64GB, KJS takes 35.7 seconds in total to verify the correctness of the algorithms associated with BST trees, and 44.8 seconds to verify correctness of the insertion sort algorithm. On a machine with an Intel Core i7-4980HQ CPU 2.80 GHz and DDR3 RAM 16GB, which is approximately 30% less powerful than the one used for KJS, JaVerT verifies the same BST algorithms in 8.5 seconds, and the insertion sort algorithm in 3.0 seconds.

2 Function Closures: Counter

JavaScript supports nested functions and function closures, meaning that one function literal can be declared within another and that functions can refer to variables that are defined in the enclosing scope. The relationships between the scope chains of such functions can become fairly complex and there are several challenges that the specification of function closures has to tackle. We illustrate these with the following example:

```
1  var make_counter = function () {
2    var count = 0;
3
4    var getCounter = function () { return count };
5    var incCounter = function () { count++ };
6    var decCounter = function () { count-- };
7
8    return { getCounter: getCounter, incCounter: incCounter, decCounter: decCounter }
9  }
10
11 var counter_1 = make_counter();
12 var counter_2 = make_counter();
13
14 counter_1.incCounter();
15 counter_1.incCounter();
16
17 counter_2.incCounter();
18
19 var count = counter_1.getCounter() + counter_2.getCounter()
```

Figure 1: Example: Function closures

The purpose of the function `make_counter` is to provide an encapsulated counter. JavaScript has no native mechanism for providing encapsulation and encapsulation is usually simulated using function closures. One declares the resources that one wishes to keep private (in our case, the variable `count`) inside the closure, and exposes only the interface for manipulating those resources (in our case, the functions `getCounter`, `incCounter`, and `decCounter`) outside the closure. In our case, this is sufficient to achieve full encapsulation—there is no way for the programmer to access the counter directly outside `make_counter`; in the general case, encapsulation cannot be guaranteed. In the rest of the program, we use `make_counter` to create two counters, we increase the first one twice, the second one once, and assign their sum to the variable `count`.

Let us specify this program. The `getCounter`, `incCounter`, and `decCounter` functions are easy—we only need to be able to talk about the variable `count`, which we do using the `scope` assertion:

```
1  /**
2    @id getCounter
3
4    @pre (
5      scope(count : #c) *
6      types(#c : $$number_type)
7    )
8
9    @post (
10     scope(count : #c) * (ret == #c)
11   )
12 */
13
14 var getCounter = function ()
15   { return count; }
```

```
1  /**
2    @id incCounter
3
4    @pre (
5      scope(count : #c) *
6      types(#c : $$number_type)
7    )
8
9    @post (
10     scope(count : #c + 1)
11   )
12 */
13
14 var incCounter = function ()
15   { count++ };
```

```
1  /**
2    @id decCounter
3
4    @pre (
5      scope(count : #c) *
6      types(#c : $$number_type)
7    )
8
9    @post (
10     scope(count : #c - 1)
11   )
12 */
13
14 var decCounter = function ()
15   { count-- };
```

The more intricate part is specifying `make_counter`. The precondition is easy, it is `emp`, and for the postcondition we need to capture the object that is returned (denoted by `c`) and all the associated resources. For starters, it is a standard JavaScript object that has three properties: `getCounter`, `incCounter`, and `decCounter`.

```
standardObject(c) *
dataField(c, "getCounter", #gc) * dataField(c, "incCounter", #ic) * dataField(c, "decCounter", #dc)
```

Next, we need to state that the values of these properties correspond to the appropriate function objects:

```
fun_obj(getCounter, #gc, #gc_sc) * fun_obj(incCounter, #ic, #ic_sc) * fun_obj(decCounter, #dc, #dc_sc)
```

The last bit of information that we need to capture is the one related to the closure—we need to express that the three functions all have the same variable `count` in their scope and that its value is equal to zero. This is done by using our `closure` abstraction:

```
closure(count: 0; getCounter: #gc_sc, incCounter: #ic_sc, decCounter: #dc_sc)
```

This assertion states that the variable `count` has value `0` in the scope chains of `getCounter`, `incCounter`, and `decCounter` (denoted by `#gc_sc`, `#ic_sc`, and `#dc_sc`), and that these particular instances of these three functions were created in the same execution of `make_counter`, i.e. that they share the same function closure.

Under the hood

The `closure` abstraction can be unfolded to show the built-in assertions that are behind it:

```
closure(v1: #v1, ..., vm: #vm; f1: #f1_sc, ..., fn: #fn_sc) :
  sc_scope(f1, v1: #v1, #f1_sc) * ... * sc_scope(f1, vm: #vm, #f1_sc) *
  o_chains(f1: #f1_sc, f2: #f2_sc) * ... o_chains(f1: #f1_sc, fn: #fn_sc) *
  o_chains(f2: #f2_sc, f3: #f3_sc) * ... o_chains(f2: #f2_sc, fn: #fn_sc) *
   ...
  o_chains(fn-1: #fn-1_sc, fn: #fn_sc)
```

The `sc_scope` assertion is a generalisation of the `scope` assertion, where `sc_scope(f, v: #v, #f_sc)` means that the variable `v` has value `#v` in the scope chain `#f_sc` of the function `f`. Recall that `scope(v : #v)` means that the variable `v` has value `#v` in the scope chain of the function being specified and the name of the function and its scope chain are left implicit.
The assertion `o_chains(f: #f_sc, g: #g_sc)`, on the other hand, is a pure assertion that states that the scope chains of the functions `f` and `g`, denoted by `#f_sc` and `#g_sc`, maximally overlap, i.e. that `f` and `g` were created in the same execution of the innermost enclosing function.

As we will be using `make_counter` to create multiple counters, it makes sense to bring these assertions together into an abstraction that captures what it means to be a counter. Its parameters should only reflect the information necessary to reason about the counter: its location and its value. In this way, we successfully abstract all of the details about the function closure.

```
pred counter(c, c_val) :
  standardObject(c) *
  dataField(c, "getCounter", #gc) *
  dataField(c, "incCounter", #ic) *
  dataField(c, "decCounter", #dc) *
  fun_obj(getCounter, #gc, #gc_sc) *
  fun_obj(incCounter, #ic, #ic_sc) *
  fun_obj(decCounter, #dc, #dc_sc) *
  closure(count: c_val; getCounter: #gc_sc, incCounter: #ic_sc, decCounter: #dc_sc) *
  types(c_val : $$number_type)
```

With this abstraction, the specification of `make_counter` becomes quite compact:

```
/**
  @id makeCounter

  @pre (emp)

  @post (counter(ret, 0))
*/
var make_counter = function () {
   var count = 0;

   var getCounter = function () { return count };
   var incCounter = function () { count++ };
   var decCounter = function () { count-- };

   return { getCounter: getCounter, incCounter: incCounter, decCounter: decCounter }
}
```

Finally, we can specify the top-level code:

```
/**
 @toprequires (emp)
 @topensures (
   scope(make_counter: #mc) *
   fun_obj(makeCounter, #mc, #mc_proto) *
   scope(counter_1: #c1) * counter(#c1, 2) *
   scope(counter_2: #c2) * counter(#c2, 1) *
   scope(count : 3)
 )
*/

var make_counter = function () {
   var count = 0;

   var getCounter = function () { return count };
   var incCounter = function () { count++ };
   var decCounter = function () { count-- };

   return { getCounter: getCounter, incCounter: incCounter, decCounter: decCounter }
}

var counter_1 = make_counter();
var counter_2 = make_counter();

counter_1.incCounter();
counter_1.incCounter();

counter_2.incCounter();

var count = counter_1.getCounter() + counter_2.getCounter()
```

The specification states that we start from an empty precondition, and finish with: the variable `make_counter`, holding the function object corresponding to the `make_counter` function; the variables `counter_1` and `counter_2`, holding counters whose values are, respectively, 2 and 1; and the variable `count`, whose value is 3.

3 Dynamic Property Access: Key-Value Map

```
/**
  @pred validKey(key) : isNamedProperty(key) * (! (key == "hasOwnProperty"));

  @pred invalidKey(key) :
    types (key : $$boolean_type), types (key : $$number_type), types (key : $$string_type) * (key == "hasOwnProperty");

  @pred Map(m, contents, mp) :
    ObjectWithProto(m, mp) * types(mp : $$object_type) *
    dataField(m, "_contents", contents) * standardObject(contents) *
    ((contents, "hasOwnProperty") -> None) * ((m, "get") -> None) * ((m, "put") -> None);

  @pred MapProto(mp) :
    standardObject(mp) * ((mp, "_contents") -> None) *
    dataField(mp, "get", #get_loc) * fun_obj(mapGet, #get_loc, #get_proto, #get_sc) *
    dataField(mp, "put", #put_loc) * fun_obj(mapPut, #put_loc, #put_proto, #put_sc);
*/

/**
  @id isValidKey

  @pre ((key == #key) * validKey(#key))           @post (ret == $$t)
  @pre ((key == #key) * invalidKey(#key))         @post (ret == $$f)
*/
function isValidKey(key) { return (typeof(key) === "string" && key !== "hasOwnProperty") }

/** @id map

  @pre (ObjectWithProto(this, #mp) * MapProto(#mp) *
    ((this, "_contents") -> None) * ((this, "get") -> None) * ((this, "put") -> None))
  @post (Map(this, #contents, #hp) * MapProto(#mp))
*/
function Map () { this._contents = {}; }

/**
  @id mapGet

  @pre (Map(this, #contents, #mp) * MapProto(#mp) * (key == #key) * validKey(#key) * dataField(#contents, #key, #v))
  @post (Map(this, #contents, #mp) * MapProto(#mp) * dataField(#contents, #key, #v) * (ret == #v))

  @pre (Map(this, #contents, #mp) * MapProto(#mp) * (key == #key) * validKey(#key) * emptyField(#contents, #key))
  @post (Map(this, #contents, #mp) * MapProto(#mp) * emptyField(#contents, #key) * (ret == $$null))

  @pre ((key == #key) * invalidKey(#key))
  @posterr (ErrorObjectWithMessage(err, "Invalid Key"))
*/
Map.prototype.get = function getValue (key) {
  if (isValidKey(key)) {
    if (this._contents.hasOwnProperty(key)) { return this._contents[key] } else { return null }
  } else { throw new Error("Invalid Key") }
}

/**
  @id mapPut

  @pre (Map(this, #contents, #mp) * MapProto(#mp) *
    (key == #key) * validKey(#key) * (value == #value) * emptyField(#contents, #key))
  @post (Map(this, #contents, #mp) * MapProto(#mp) * dataField(#contents, #key, #value))

  @pre (Map(this, #contents, #mp) * MapProto(#mp) *
    (key == #key) * validKey(#key) * (value == #value) * dataFieldGeneral(#contents, #key, #oV, $$t, #e, #c))
  @post (Map(this, #contents, #mp) * MapProto(#mp) * dataFieldGeneral(#contents, #key, #value, $$t, #e, #c))

  @pre (Map(this, #contents, #mp) * MapProto(#mp) *
    (key == #key) * validKey(#key) * (value == #value) * dataFieldGeneral(#contents, #key, #oV, $$f, #e, #c))
  @posterr (Map(this, #contents, #mp) * MapProto(#mp) * dataFieldGeneral(#contents, #key, #oV, $$f, #e, #c) *
    isTypeError(err))

  @pre ((key == #key) * invalidKey(#key))
  @posterr (ErrorObjectWithMessage(err, "Invalid Key"))
*/
Map.prototype.put = function (key, value) {
   if (isValidKey(key)) { this._contents[key] = value } else throw new Error("Invalid Key")
}
```

We illustrate our handling of dynamic property access by appealing to the example of a key-value map implementation in JavaScript, shown and specified in full on the previous page. A map is an object whose property `_contents` holds an object that serves as the actual map—its property names (which are strings) are the map keys and their values (which are arbitrary) are the map values. We provide functionalities for retrieving and setting the value of a given key (`get` and `put`), both of which we store in `Map.prototype`.

This example draws several parallels with the priority queue example of the paper. However, reasoning about contents of a map has an additional layer of complexity. While node objects always have the static `pri`, `val`, and `next` properties inside node, the objects representing the contents of maps may have arbitrary properties. This is seen in the specifications of `get` and `put` functions; more specifically, in the abstractions `dataField(#contents, #key)` and `emptyField(#contents, #key)`, where the value of `#key` is not staticall known. JSIL Verify was designed to handle dynamic property access and has no problems reasoning about it, as indicated by the time required to verify all of the specifications shown in Table 1.

This example illustrates the common practice of testing if an object has an own property using the built-in function `hasOwnProperty`, which lives in `Object.prototype`. Specifying this example reveals a potential flaw in such programs, that has to do with prototype safety. Namely, the hashtable can override `hasOwnProperty` by having the key `"hasOwnProperty"` itself, and again, the specifications have to account for that. Our ad-hoc solution is to exclude the string `hasOwnProperty` from the set of valid keys. There are different ways in which this can be handle more elegantly, but these are not the point of this example.

Further work

This specification can be written more abstractly. We could have a `Map` abstraction capturing the property names currently in the map, together with their values. This abstraction, together with similar abstractions that would allow us to reason about, for example, the JavaScript `for−in` construct, are part of our immediate further work.