traits.js

Robust Object Composition and High-integrity Objects for ECMAScript 5

Tom Van Cutsem
Software Languages Lab
Vrije Universiteit Brussel, Belgium
tvcutsem@vub.ac.be

Mark S. Miller
Google, USA
erights@google.com

Abstract

This paper introduces `traits.js`, a small, portable trait composition library for Javascript. Traits are a more robust alternative to multiple inheritance and enable object composition and reuse. `traits.js` is motivated by two goals: first, it is an experiment in using and extending Javascript's recently added meta-level object description format. By reusing this standard description format, `traits.js` can be made more interoperable with similar libraries, and even with built-in primitives. Second, `traits.js` makes it convenient to create "high-integrity" objects whose integrity cannot be violated by clients, an important property in the context of interaction between mutually suspicious scripts.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Object-oriented languages

General Terms Design, Languages

Keywords Traits, Javascript

1. Introduction

We introduce `traits.js`, a small, standards-compliant trait composition library for ECMAScript 5, the latest standard of Javascript. Traits are a more robust alternative to classes with multiple inheritance.

A common pattern in Javascript is to add ("mixin") the properties of one object to another object. `traits.js` provides a few simple functions for performing this pattern safely as it will detect, propagate and report conflicts (name clashes) created during a composition. While such a library is certainly useful, it is by no means novel. Because of Javascript's flexible yet low-level object model, libraries that add class-like abstractions with mixin- or trait-like capabilities abound. What sets `traits.js` apart?

Standard object representation format `traits.js` represents traits in terms of a new meta-level object description format, introduced in the latest ECMAScript 5th edition (ES5) [3]. The use of such a standard format, rather than inventing an ad hoc representation, allows higher interoperability with other libraries that use this format, including the built-in functions defined by ES5 itself. We briefly describe ES5's new object-description API in the following Section. We show how this standard object description format lends itself well to extensions of Javascript object semantics, while remaining interoperable with other libraries.

Support for high integrity `traits.js` facilitates the creation of so-called "high-integrity" objects. By default, Javascript objects are extremely dynamic: clients can add, remove and assign to any property, and are even allowed to rebind the `this` pseudovariable in an object's methods to arbitrary other objects. While this flexibility is often an asset, in the context of cooperation between untrusted scripts it is a liability. ECMAScript 5 introduces a number of primitives that enable high-integrity objects, yet not at all in a convenient manner. An explicit goal of `traits.js` is to make it as convenient to create high-integrity objects as it is to create Javascript's standard, dynamic objects.

Minimal `traits.js` introduces just the necessary features to create, combine and instantiate traits. It does not add the concept of a class to Javascript, but rather reuses Javascript functions for the roles traditionally attributed to classes. Inspired by the first author's earlier work [7], a class in this library is just a function that returns new trait instances.

Availability `traits.js` can be downloaded from `www.traitsjs.org` and runs in all major browsers. It also runs in server-side Javascript environments, like `node.js`.

2. ECMAScript 5

Before introducing traits.js proper, we briefly touch upon a number of features introduced in the most recent version of ECMAScript. Understanding these features is key to understanding traits.js.

Property Descriptors ECMAScript 5 defines a new object-manipulation API that provides more fine-grained control over the nature of object properties [3]. In Javascript, objects are records of *properties* mapping names (strings) to values. A simple two-dimensional point whose y-coordinate always equals the x-coordinate can be defined as:

```
var point = {
  x: 5,
  get y() { return this.x; },
  toString: function() { return '[Point '+this.x+']'; }
};
```

ECMAScript 5 distinguishes between two kinds of properties. Here, x is a *data property*, mapping a name to a value directly. y is an *accessor property*, mapping a name to a "getter" and/or a "setter" function. The expression point.y implicitly calls the getter function.

ECMAScript 5 further associates with each property a set of *attributes*. Attributes are meta-data that describe whether the property is writable (can be assigned to), enumerable (whether it appears in for-in loops) or configurable (whether the property can be deleted and whether its attributes can be modified). The following code snippet shows how these attributes can be inspected and defined:

```
var pd = Object.getOwnPropertyDescriptor(o, 'x');
// pd = {
//    value: 5,
//    writable : true,
//    enumerable: true,
//    configurable : true
// }
Object.defineProperty(o, 'z', {
  get: function() { return this.x; },
  enumerable: false,
  configurable: true
});
```

The pd object and the third argument to defineProperty are called *property descriptors*. These are objects that describe properties of objects. Data property descriptors declare a value and a writable property, while accessor property descriptors declare a get and/or a set property.

The Object.create function can be used to generate new objects based on a set of property descriptors directly. Its first argument specifies the prototype of the object to be created (every Javascript object forwards requests for properties it does not know to its prototype). Its second argument is an object mapping property names to property descriptors. This object, which we will refer to as a *property descriptor map*, describes both the properties and the meta-data (writability, enumerability, configurability) of the object to be created. Armed with this knowledge, we could have also defined the point object explicitly as:

```
var point = Object.create(Object.prototype, {
  x: { value: 5,
       enumerable: true,
       writable: true,
       configurable: true },
  y: { get: function() { return this.x; },
       enumerable: true,
       configurable: true },
  toString: { value: function()  {...},
              enumerable: true,
              writable: true,
              configurable: true }
});
```

Tamper-proof Objects ECMAScript 5 supports the creation of tamper-proof objects that can protect themselves from modifications by client objects. Objects can be made *non-extensible*, *sealed* or *frozen*. A non-extensible object cannot be extended with new properties. A sealed object is a non-extensible object whose own (non-inherited) properties are all non-configurable. Finally, a frozen object is a sealed object whose own properties are all non-writable. The call Object.freeze(obj) freezes the object obj. As we will describe in Section 6, traits.js supports the creation of such tamper-proof objects.

Bind A common pitfall in Javascript relates to the peculiar binding rules for the this pseudovariable in methods [2]. For example:

```
var obj = {
  x:1,
  m: function() { return this.x; }
};
var meth = obj.m; // grab the method as a function
meth(); // "this" is now set to the global object
```

Javascript methods are simply functions stored in objects. When calling a method obj.m(), the method's this pseudovariable is bound to obj, as expected. However, when accessing a method as a property obj.m and storing it in a variable meth, as is done in the above example, the function loses track of its this-binding. When it is subsequently called as meth(), this is bound to the global object by default, returning the wrong value for this.x.

There are other ways for the value of this to be rebound. Any object can call a method with an explicit binding for this, by invoking meth.call(obj). While that solves the problem in this case, unfortunately, in general, malicious clients can use the call primitive to confuse the original method by binding its this pseudovariable to a totally unrelated object. To guard against such this-rebinding, whether by accident or by intent, one can use the ECMAScript 5 bind method, as follows:

```
obj.m = obj.m.bind(obj); // fixes m's "this" to "obj"
var meth = obj.m;
meth(); // returns 1 as expected
```

Now m can be selected from the object and passed around as a function, without fear of accidentally having its this rebound to the global object, or any other random object.

3. Traits

Traits were originally defined as "composable units of behavior" [5]: reusable groups of methods that can be composed together to form a class. Trait composition can be thought of as a more robust alternative to multiple inheritance. Traits may provide and require a number of methods. Required methods are like abstract methods in OO class hierarchies: their implementation should be provided by another trait or class.

The main difference between traits and alternative composition techniques such as multiple inheritance and mixin-based inheritance [1] is that upon trait composition, name conflicts (a.k.a. name clashes) should be explicitly resolved by the composer. This is in contrast to multiple inheritance and mixins, which define various kinds of linearization schemes that impose an implicit precedence on the composed entities, with one entity overriding all of the methods of another entity. While such systems often work well in small reuse scenarios, they are not robust: small changes in the ordering of classes/mixins somewhere high up in the inheritance/mixin chain may impact the way name clashes are resolved further down the inheritance/mixin chain [6]. In addition, the linearization imposed by multiple inheritance or mixins precludes a composer to give precedence to both a method m1 from one class/mixin A and a method m2 from another class/mixin B: either all of A's methods take precedence over B, or all of B's methods take precedence over A.

Traits allow a composing entity to resolve name clashes in the individual components by either excluding a method from one of the components or by having one trait explicitly override the methods of another one. In addition, the composer may define an alias for a method, allowing the composer to refer to the original method even if its original name was excluded or overridden.

Name clashes that are never explicitly resolved will eventually lead to a composition error. Depending on the language, this composition error may be a compile-time error, a runtime error when the trait is composed, or a runtime error when a conflicting name is invoked on a trait instance.

Trait composition is declarative in the sense that the ordering of composed traits does not matter. In other words, unlike mixin-based or multiple inheritance, trait composition is commutative and associative. This tremendously reduces the cognitive burden of reasoning about deeply nested levels of trait composition. In languages that support traits as a compile-time entity (similar to classes), trait composition can be entirely performed at compile-time, effectively "flattening" the composition and eliminating any composition overhead at runtime.

Since their publication in 2003, traits have received widespread adoption in the PL community, although the details of the many traits implementations differ significantly from the original implementation defined for Smalltalk. Traits have been adopted in a.o. Perl, Fortress and Scheme [4].

4. traits.js in a Nutshell

As a concrete example of a trait, consider the "enumerability" of collection objects. In many languages, collection objects all support a similar set of methods to manipulate the objects contained in the collection. Most of these methods are generic across all collections and can be implemented in terms of just a few collection-specific methods, e.g. a method forEach that returns successive elements of the collection. Such a TEnumerable trait can be encoded using traits.js as follows:

```
var TEnumerable = Trait({
  // required property, to be provided later
  forEach: Trait.required,
  // provided properties
  map: function(fun) {
    var r = [];
    this.forEach(function (e) { r.push(fun(e)); });
    return r;
  },
  reduce: function(init, accum) {
    var r = init;
    this.forEach(function (e) { r = accum(r,e); });
    return r;
  },
   ...
});

// an example enumerable collection
function Range(from, to) {
  return Trait.create(
    Object.prototype,
    Trait.compose(
      TEnumerable,
      Trait({
        forEach: function(fun) {
          for (var i = from; i < to; i++) { fun(i); }
        }
       })));
}
```

```
var r = Range(0,5);
r.reduce(0, function(a,b){return a+b;}); // 10
```

traits.js exports a single function object, named Trait. Calling Trait({...}) creates and returns a new trait. We refer to this Trait function as the Trait constructor. The Trait constructor additionally defines a number of properties:

- Trait.required is a special singleton value that is used to denote missing required properties. traits.js recognizes such data properties as required properties and they are treated specially by Trait.create and by Trait.compose (as explained later). Traits are not required to state their required properties explicitly, but it is often useful to do so for documentation purposes.
- The function Trait.compose takes an arbitrary number of input traits and returns a composite trait.
- The function Trait.create takes a prototype object and a trait, and returns a new trait instance. The first argument is the prototype of the trait instance. Note the similarity to the built-in Object.create function.

When a trait is instantiated into an object o, the binding of the this pseudovariable of the trait's methods refers to o. In the example, the TEnumerable trait defines two methods, map and reduce, that require (depend on) the forEach method. This dependency is expressed via the self-send this.forEach(...). When map or reduce is invoked on the fully composed Range instance r, this will refer to r, and this.forEach refers to the method defined in the Range function.

5. Traits as Property Descriptor Maps

We now describe the unique feature of traits.js, namely the way in which it represents trait objects. traits.js represents traits as property descriptor maps (cf. Section 2): objects whose keys represent property names and whose values are property descriptors. Hence, traits conform to an "open" representation, and are not opaque values that can only be manipulated by the functions exported by the library. Quite the contrary: by building upon the property descriptor map format, libraries that operate on property descriptors can also operate on traits, and the traits.js library can consume property descriptor maps that were not constructed by the library itself.

Figure 1 depicts the different kinds of objects that play a role in traits.js and the conversion functions between them. These conversions are explained in more detail in the following Sections.

5.1 Simple (non-composite) Traits

Recall that the Trait function acts as a constructor for simple (non-composite) traits. It essentially turns an object describing a record of properties into a trait. For example:

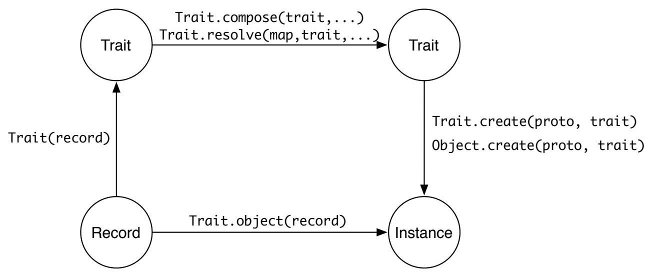
Trait
Trait.compose(trait,...)
Trait.resolve(map,trait,...)
Trait
Trait(record)
Trait.create(proto, trait)
Object.create(proto, trait)
Record
Trait.object(record)
Instance

Figure 1. Object types and conversions in traits.js

```
var T = Trait({
  a: Trait.required,
  b: "foo",
  c: function() { ... }
});
```

The above trait T provides the properties b and c and requires the property a. The Trait constructor converts the object literal into the following property descriptor map T, which represents a trait:

```
{ 'a' : {
    value: undefined,
    required: true,
    enumerable: false,
    configurable: false
  },
  'b' : {
    value: "foo",
    writable: false,
    enumerable: true,
    configurable: false
  },
  'c' : {
    value: function() { ... },
    method: true,
    enumerable: true,
    configurable: false
  }
}
```

The attributes required and method are not standard ES5 attributes, but are recognized and interpreted by the Trait.create function described later.

The objects passed to Trait are meant to serve as plain *records* that describe a simple trait's properties. Just like Javascript itself has a convenient and short object literal syntax, in addition to the more heavyweight, yet more powerful Object.create syntax (as shown in Section 2), passing a record to the Trait constructor is a handy way of defining a trait without having to spell out all meta-data by hand.

The Trait function turns a record into a property descriptor map with the following constraints:

- Only the record's own properties are turned into trait properties (its prototype is not significant, inherited properties are ignored).
- Data properties in the record bound to the special `Trait.required` singleton are bound to a property descriptor marked with the `required: true` attribute.
- Data properties in the record bound to functions are marked with the `method: true` attribute. `traits.js` distinguishes between such methods and plain function-valued data properties in the following ways:
  - Normal Javascript functions are mutable objects, but trait methods are treated as frozen objects (i.e. objects with immutable structure).
  - For normal Javascript functions, their `this` pseudovariable is a free variable that can be set to any object by callers. For trait methods, the `this` pseudovariable of a method will be bound to trait instances, disallowing callers to specify a different value for `this`.

5.2 Composing Traits

The function `Trait.compose` is the workhorse of `traits.js`. It composes zero or more traits into a single composite trait:

```
var T1 = Trait({ a: 0, b: 1});
var T2 = Trait({ a: 1, c: 2});
var Tc = Trait.compose(T1,T2);
```

The composite trait contains the union of all properties from the argument traits. For properties whose name appears in multiple argument traits, a distinct "conflicting" property is defined in the composite trait. The format of `Tc` is:

```
{ 'a' : {
    get: function(){ throw ...; },
    set: function(){ throw ...; },
    conflict: true
  },
  'b' : { value: 1 },
  'c' : { value: 2 } }
```

The conflicting `a` property in the composite trait is marked as a conflicting property by means of a `conflict: true` attribute (again, this is not a standard ES5 attribute). Conflicting properties are accessor properties whose `get` and `set` functions raise an appropriate runtime exception when invoked.

Two properties `p1` and `p2` with the same name are not in conflict if:

- `p1` or `p2` is a required property. If either `p1` or `p2` is a non-required property, the required property is overridden by the non-required property.
- `p1` and `p2` denote the same property. Two properties are considered to be the same if they refer to identical values and have identical attribute values. This implies that it is OK for the same property to be "inherited" via different composition paths, e.g. in the case of diamond inheritance.

`compose` is a commutative and associative operation: the ordering of its arguments does not matter, and `compose(t1,t2,t3)` is equivalent to `compose(t1,compose(t2,t3))` or `compose(compose(t2,t1),t3)`.

5.3 Resolving Conflicts

The `Trait.resolve` function can be used to resolve conflicts created by `Trait.compose`, by either renaming or excluding conflicting property names. The function takes as its first argument an object that maps property names to either strings (indicating that the property should be renamed) or to `undefined` (indicating that the property should be excluded). `Trait.resolve` returns a fresh trait in which the indicated properties have been renamed or excluded.

For example, if we wanted to avoid the conflict in the `Tc` trait from the previous example, we could have composed `T1` and `T2` as follows:

```
var Trenamed =
  Trait.compose(T1, Trait.resolve({ a: 'd' }, T2);
var Texclude =
  Trait.compose(T1, Trait.resolve({ a: undefined }, T2);
```

`Trenamed` and `Texclude` have the following structure:

```
// Trenamed =
{ 'a' : { value: 0 },
  'b' : { value: 1 },
  'c' : { value: 2 },
  'd' : { value: 1 } } // T2.a renamed to 'd'
// Texclude =
{ 'a' : { value: 0 },   // T2.a excluded
  'b' : { value: 1 },
  'c' : { value: 2 } }
```

When a property `p` is renamed or excluded, `p` itself is turned into a required property, to attest that the trait is not valid unless the composer provides an alternative implementation for the old name.

5.4 Instantiating Traits

`traits.js` provides two ways to instantiate a trait: using its own provided `Trait.create` function, or using the ES5 `Object.create` primitive. We discuss each of these below.

Trait.create When instantiating a trait, `Trait.create` performs two "conformance checks". A call to `Trait.create(proto, trait)` fails if:

- `trait` still contains required properties, and those properties are not provided by `proto`. This is analogous to trying to instantiate an abstract class.

- `trait` still contains conflicting properties.

In addition, `traits.js` ensures that the new trait instance has high integrity:

- The `this` pseudovariable of all trait methods is bound to the new instance, using the `bind` method introduced in Section 2. This ensures clients cannot tamper with a trait instance's `this`-binding.
- The instance is created as a *frozen* object: clients cannot add, delete or assign to the instance's properties.

Object.create Since `Object.create` is an ES5 built-in that knows nothing about traits, it will not perform the above trait conformance checks and will not fail on incomplete or inconsistent traits. Instead, required and conflicting properties are interpreted as follows:

- Required properties will be bound to `undefined`, and will be non-enumerable (i.e. they will not show up in `for-in` loops on the trait instance). This makes such properties virtually invisible (in Javascript, if an object `o` does not define a property `x`, `o.x` also returns `undefined`). Clients can still assign a value to these properties later.
- Conflicting properties have a getter and a setter that throws an exception when accessed. Hence, the moment a program touches a conflicting property, it will fail, revealing the unresolved conflict.

`Object.create` does not bind `this` for trait methods and does not generate frozen instances. Hence, the new trait instance can still be modified by clients.

It is up to the programmer to decide which instantiation method, `Trait.create` or `Object.create`, is more appropriate: `Trait.create` fails on incomplete or inconsistent traits and generates frozen objects, `Object.create` may generate incomplete or inconsistent objects, but as long as a program never actually touches a conflicting property, it will work fine (which fits with the dynamically typed nature of Javascript).

In summary, because `traits.js` reuses the ES5 property descriptor format to represent traits, it interoperates well with libraries that operate on the same format, including the built-in primitives. While such libraries do not understand the additional attributes used by `traits.js` (such as `required:true`), sometimes it is still possible to encode the semantics of those attributes by means of the standard attributes. By carefully choosing the representation for required and conflicting properties, we were able to have `Object.create` behave reasonably for traits. Furthermore, the semantics provided by `Object.create` provide a nice alternative to the semantics provided by `Trait.create`: the former provides dynamic, late error checks and generates flexible instances, while the latter provides early error checks and generates high-integrity instances.

6. High-integrity Objects

In Section 2 we mentioned that ECMAScript 5 supports tamper-proof objects by means of three new primitives that can make an object non-extensible, sealed or frozen. At first sight, these primitives seem sufficient to construct high-integrity objects, that is: objects whose structure or methods cannot be changed by client objects. While freezing an object fixes its structure, it does not fix the `this`-binding issue for methods, and leaves methods as fully mutable objects. Hence, simply calling `Object.freeze(obj)` does not produce a high-integrity object.

`traits.js`, by means of its `Trait.create` function, provides a more convenient alternative to construct high-integrity objects: a trait instance constructed by this function is frozen and has frozen methods whose `this` pseudovariable is fixed to the trait instance using `bind`.

In order to construct the 2D point object from Section 2 as a high-integrity object in plain ECMAScript 5, one has to write approximately[1] the following:

```
var point = {
  x: 5,
  toString: function() { return '[Point '+this.x+']'; }
};
point.toString =
  Object.freeze(point.toString.bind(point));
Object.defineProperty(point, 'y', {
  get: Object.freeze(
    function() { return this.x; }).bind(point)
});
Object.freeze(point);
```

With `traits.js`, the above code can be simplified to:

```
var point = Trait.create(Object.prototype,
  Trait({
    x: 5,
    get y() { return this.x; },
    toString: function() { return '[Point '+this.x+']'; }
  }));
```

In the above example, the original code for `point` was wrapped in a `Trait` constructor. This trait is then immediately instantiated using `Trait.create` to produce a high-integrity object. To better support this idiom, `traits.js` defines a `Trait.object` function that combines trait declaration and instantiation, such that the example can be further simplified to:

```
var point = Trait.object({
  x: 5,
  get y() { return this.x; },
  toString: function() { return '[Point '+this.x+']'; }
});
```

[1] To fully fix the object's structure, the prototype of its methods should also be fixed.

This pattern makes it feasible to work with high-integrity objects by default.

7. Library or Language Extension?

Traits are not normally thought of as a library feature, but rather as a declarative language feature, tightly integrated with the language semantics. By contrast, `traits.js` is a stand-alone Javascript library. We found that `traits.js` is quite pleasant to use as a library without dedicated syntax.

Nevertheless, there are issues with traits as a library, especially with the design of `traits.js`. In particular, binding the `this` pseudovariable of trait methods to the trait instance, to prevent `this` from being set by callers, requires a bound method wrapper per method per instance. Hence, instances of the same trait cannot share their methods, but rather have their own per-instance wrappers. This is much less efficient than the method sharing afforded by Javascript's built-in prototypal inheritance.

We did design `traits.js` in such a way that a smart Javascript engine could partially evaluate trait composition statically, provided that the library is used in a restricted manner. If the argument to `Trait` is an object literal rather than an arbitrary expression, then transformations like the one below apply:

```
Trait.compose(Trait({ a: 1 }), Trait({ b: 2}))
−>
Trait({ a:1, b:2 })
```

Transformations like these would not only remove the runtime cost of trait composition, they would also enable implementations to recognize calls to `Trait.create` that generate instances of a single kind of trait, and replace those calls to specialized versions of `Trait.create` that are partially evaluated with the static trait description. The implementation can then make sure that all trait instances generated by this specialized method efficiently share their common structure.

Because of the dynamic nature of Javascript, and the brittle usage restrictions required to enable the transformations, the cost of reliably performing the sketched transformations is high. An extension of Javascript with proper syntax for trait composition would obviate the need for such complex optimizations, and would likely improve error reporting and overall usability as well.

8. Micro-benchmarks

This section reports on a number of micro-benchmarks that try to give a feel for the overhead of `traits.js` as compared to built-in Javascript object creation and method invocation.

The results presented here were obtained on an Intel Core 2 Duo 2.4Ghz Macbook with 4GB of memory, running Mac OS X 10.6.8 and using the Javascript engines of three modern web browsers, with the latest `traits.js` version 0.4. In the interest of reproducibility, the source code of the microbenchmarks used here is available at `http://es-lab.googlecode.com/files/traitsjs-microbench.html`.

First, *independent of `traits.js`*, we note that creating an object using the built-in `Object.create` function is easily a factor of 10 slower than creating objects via the standard prototypal inheritance pattern, whereby an object is instantiated by calling `new` on a function, and methods are stored in the object's prototype, rather than in the object directly.

Therefore, in Table 1, we compare the overhead of `traits.js` relative to creating an object using the built-in `Object.create` API. The numbers shown are the ratios between runtimes. Each number is the mean ratio of 5 runs (each in an independent, sufficiently warmed-up browser session), including the standard deviation from the mean.

The first three rows report the overhead of *allocating* a new trait instance with respectively 10, 100 or 1000 methods, compared to allocating a non-trait object with an equal amount of methods (using `Object.create`). The column indicates whether the trait instance was created using `Trait.create` or `Object.create`[2].

Across different platforms and sizes, there is roughly a factor of 10 slowdown when using `Trait.create`. This overhead stems from both additional trait conformance checks (checks for missing required and remaining conflicting properties), and the creation of bound methods. As expected, there is no particular overhead when instantiating traits using `Object.create`. Bear in mind that `Object.create` itself is easily 10x slower than prototypal object creation.

The last row measures the overhead of invoking a method on a trait instance, compared to invoking a method on a regular object. Since `Trait.create` creates bound methods, there is a 1.56 to 3.93x slowdown compared to a standard method invocation. Again, for instances created by `Object.create` there is no overhead, since such instances do not have bound methods.

In closing, note that these micro-benchmarks do not in any way inform us of the actual overhead of `traits.js` in a realistic Javascript application.

9. Conclusion

`traits.js` is a small, standards-compliant trait composition library for Javascript. The novelty of `traits.js` is that it uses a standard object-description format, introduced in the recent ECMAScript 5 standard, to represent traits. Traits are not opaque values but an open set of property descriptors. This increases interoperability with other libraries using the same format, including built-in primitives.

By carefully choosing the representation of traits in terms of property descriptor maps, `traits.js` allows traits to be instantiated in two ways: using its own library-provided

[2] On Chrome, for traits of size 1000, we achieved unreliable results due to excessive slowdowns. Those results are excluded from the table.

allocation	**Firefox 7.0.1**		**Chrome 14.0.835.202**		**Safari 5.1 (6534.50)**	
	Trait.create	Object.create	Trait.create	Object.create	Trait.create	Object.create
size 10	9.36x ±.48	1.05x ±.06	10.48x ±2.92	0.79x ±.20	11.45x ±.78	1.00x ±.00
size 100	10.80x±.28	.99x ±.01	9.72x ±0.36	1.02x ±.05	8.28x ±.28	1.11x ±.06
size 1000	10.26x±.58	.97x ±.04			7.77x ±.40	.98x ±.02
method call	1.56x ±.07	1.00x ±.04	3.93x ±.88	.80x ±.16	1.92x ±.18	1.00x ±.00

Table 1. Overhead of `traits.js` versus built-in `Object.create`.

function, `Trait.create`, which performs early conformance checks and produces high-integrity instances; or using the ES5 `Object.create` function, which is oblivious to any trait semantics, yet produces meaningful instances with late, dynamic conformance checks. This freedom of choice allows `traits.js` to be used both in situations where high-integrity and extensibility are required.

Finally, the convenience afforded by `Trait.object` makes it feasible to work with high-integrity objects by default. We feel this is an important addition to the Javascript programmer's toolbox.

Acknowledgments

We thank the anonymous referees, the members of the ECMAScript committee and the `es-discuss` mailing list for their valuable feedback.

Tom Van Cutsem is a Postdoctoral Fellow of the Research Foundation, Flanders (FWO). Part of this work was carried out while the first author was on a Visiting Faculty appointment at Google, sponsored by Google and a travel grant from the FWO.

References

[1] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90*, pages 303–311, New York, NY, USA, 1990. ACM.

[2] D. Crockford. *Javascript: The Good Parts*. O'Reilly, 2008.

[3] ECMA International. *ECMA-262: ECMAScript Language Specification*. ECMA, Geneva, Switzerland, fifth edition, December 2009.

[4] M. Flatt, R. B. Finder, and M. Felleisen. Scheme with classes, mixins and traits. In *AAPLAS '06*, 2006.

[5] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP '03*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[6] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86*, pages 38–45, New York, NY, USA, 1986. ACM.

[7] T. Van Cutsem, A. Bergel, S. Ducasse, and W. Meuter. Adding state and visibility control to traits using lexical nesting. In *ECOOP '09*, pages 220–243, Berlin, Heidelberg, 2009. Springer-Verlag.