Robust Trait Composition for Javascript[☆]

Tom Van Cutsem[a], Mark S. Miller[b]

[a]*Software Languages Lab, Vrije Universiteit Brussel, Belgium*
[b]*Google, USA*

Abstract

We introduce `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 when web content is composed from mutually suspicious scripts. We describe the design of `traits.js` and provide an operational semantics for TRAITS-JS, a minimal calculus that models the core functionality of the library.

Keywords: Traits, Mixins, Javascript, ECMAScript 5

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 (e.g. Prototype's `Class.create`, jQuery's `jQuery.extend`, MooTools mixins, YUI's `Y.augment`, Dojo's `dojo.mixin`, ...). What sets `traits.js` apart?

Traits, not mixins. Most of the aforementioned libraries provide support for mixins, not traits. They mostly build upon Javascript's flexible object

[☆]To appear in Science of Computer Programming, Special Issue on Advances in Dynamic Languages, Elsevier. Submitted March 2012, Revised Sept. 2012, Accepted Nov. 2012.

model which allows objects to be augmented at runtime with new properties. However, if the target of a mixin operation already contains a method that is mixed-in, the existing method is mostly simply overridden. As explained in section 3, traits support explicit conflict resolution to make object composition more robust.

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) [1]. 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 instances. `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 [2], a class in this library is just a function that returns new trait instances.

This work is an extension of our earlier paper on `traits.js` [3]. This paper adds a calculus and accompanying operational semantics that precisely captures the semantics of trait composition in `traits.js` (cf. Section 7).

Availability. `traits.js` can be downloaded from `www.traitsjs.org` and runs in all major browsers and 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 [1]. 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(point, 'x');
// pd = {
//   value: 5,
//   writable : true,
//   enumerable: true,
//   configurable : true
// }
Object.defineProperty(point, '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:

```
// Object.prototype is the 'root' prototype
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 [4]. For example:

```
var obj = {
  x:1,
  m: function() { return this.x; }
};
var meth = obj.m; // grab the method as a function
meth(); // 'this' keyword will refer 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−binding 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, 6]: 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 [7] 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 [8]. 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 composer to resolve name clashes in the combined components by either excluding a method from all but one of the components or by explicitly choosing the method of one of the components, thus implicitly overriding the other components' method. 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 other languages, although the details of the many traits implementations differ significantly from the original implementation defined for Smalltalk. Traits have been adopted in among others PHP, Perl, Fortress and Racket [9]. Although originally designed in a dynamically typed setting, several type systems have been built for Traits [10, 11, 12, 13].

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 by trait composer
  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[1]. 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 (an object). 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.

[1]Alternatively one may call `new Trait({...})`. The `new` keyword is optional in this case.

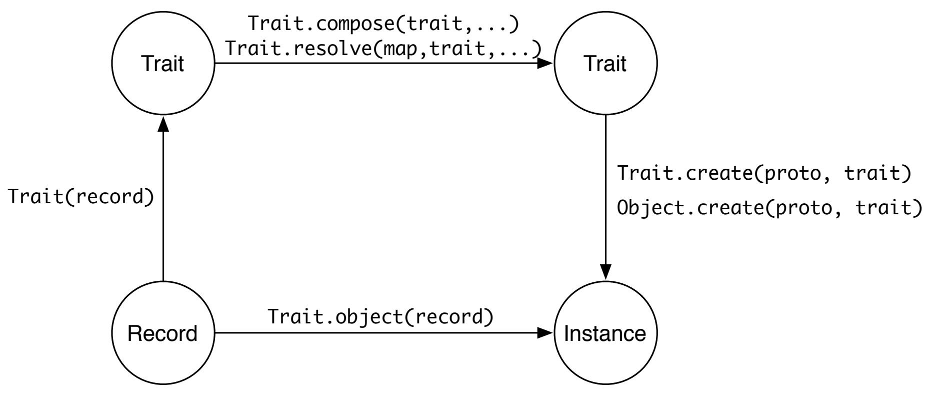
Trait
Trait.compose(trait,...)
Trait.resolve(map,trait,...)
Trait
Trait(record)
Trait.create(proto, trait)
Object.create(proto, trait)
Record
Trait.object(record)
Instance

Figure 1: Object types and conversions in `traits.js`

5.1. Atomic (non-composite) Traits

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

```
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 an atomic 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, inherited properties are ignored. This is because the prototype of the record is not significant. In Javascript, object literals by default inherit from the root `Object.prototype` object. Properties of this shared root object should not become part of the trait.
- 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`.

  The rationale for treating methods in this way will become clear in Section 6.

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 (for instance,representing required properties as non-enumerable and conflicting properties as accessors that throw). As such, even when a trait is created using `Object.create`, required and conflicting properties have a reasonable representation. 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

Recall from the introduction that one of the goals of `traits.js` is to facilitate the creation of high-integrity objects in Javascript, that is: objects whose structure or methods cannot be changed by client objects, so that a client can only properly interact with an object by invoking its methods or reading its properties.

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. Armed with these primitives, it seems that ECMAScript 5 already has good support for constructing high-integrity objects. However, while freezing an object fixes its structure, it does not fix the `this`-binding issue for methods, and leaves methods as fully mutable objects, still making it possible for clients to tamper with them. 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 robust 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`. A client can only use a trait instance by invoking its methods or reading its fields.

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[2] 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+']'; }
});
```

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

7. Operational Semantics

The goal of this section is to provide a precise description of the key functionality of the `traits.js` library. In particular, we want to formally specify:

- the `this`-binding of methods in the presence of method extraction (extracting a method as a first-class function) and method binding (as performed by `Trait.create`).
- the distinction between trait instances generated by `Trait.create` and `Object.create`.

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

- the precise rules of trait composition and renaming.

As Javascript is a complex language to formalize, we introduce the TRAITS-JS calculus, which models a simple subset of a Javascript-like language with built-in support for traits. TRAITS-JS features objects with data and method properties. As in Javascript, method properties can be extracted from an object as first-class functions. TRAITS-JS does not model Javascript's prototypal inheritance.

As in `traits.js`, traits can either be defined as atomic traits or by composing existing traits via the `compose`, `override` and `resolve` operators. Objects in TRAITS-JS can be instantiated from traits only, either via the `newTrait` operator (which models the `Trait.create` function) or via the `newObject` operator (which models the `Object.create` function). While traits in TRAITS-JS are first-class, they are not represented as objects, to clearly separate trait declarations from trait instances.

Figure 2 depicts the values of a TRAITS-JS program. Trait declarations are modelled as a data type distinct from objects (trait instances). Both trait declarations and objects are represented by a partial function $s \mapsto p$ mapping strings s to properties p. Objects are additionally represented by a boolean flag b that models whether or not the object is extensible. First-class functions are represented by their parameter list $\overline{x}$, their method body e and a reference to a bound `this` object v. v can be `null`, in which case the function is unbound. Contrary to Javascript, functions in TRAITS-JS are not full objects with their own properties. They are first-class, but the only useful operation supported on them is invocation.

$$\begin{array}{rcll}
o, t \in \textbf{Object} & ::= & \mathcal{O}\langle s \mapsto p, b\rangle & \text{Objects} \\
 & \mid & \mathcal{T}\langle s \mapsto p\rangle & \text{Traits} \\
 & \mid & \mathcal{F}\langle \overline{x}, e, v\rangle & \text{Functions} \\
p \in \textbf{Property} & ::= & \mathcal{D}\langle v, b\rangle & \text{Data property} \\
 & \mid & \mathcal{M}\langle \overline{x}, e\rangle & \text{Method property} \\
 & \mid & \mathcal{R} & \text{Required property} \\
 & \mid & \mathcal{C} & \text{Conflicting property} \\
v \in \textbf{Value} & ::= & \iota_o \mid \text{null} & \text{Values} \\
H \in \textbf{Heap} & ::= & \textbf{ObjectId} \mapsto \textbf{Object} & \text{Heaps}
\end{array}$$

$$\iota_o \in \textbf{ObjectId}, x \in \textbf{VarName}, b \in \textbf{Boolean}, s \in \textbf{String}$$

Figure 2: Semantic entities of TRAITS-JS.

Trait declarations consist of four kinds of properties. Data properties $\mathcal{D}\langle v, b\rangle$ are tuples of a value v and a boolean flag b, indicating whether the property binding is frozen (i.e. constant). Method properties $\mathcal{M}\langle \overline{x}, e\rangle$ store the formal parameters $\overline{x}$ and method body e of trait methods. Method properties of trait instances are immutable (i.e. cannot be updated). Required properties $\mathcal{R}$ mark property names to be provided by other traits. Conflicting properties $\mathcal{C}$ mark conflicting property names.

Values v are either references to objects, traits or functions, or the `null` value. Finally, the heap H is a partial function from such references to the actual object, trait or function representations.

7.1. Syntax

TRAITS-JS defines standard expressions for referring to and introducing lexically scoped variables x. These variable bindings are immutable. There are standard expressions for accessing, updating and invoking the properties of an object.

Atomic traits are defined using the syntax `trait{...}`. Data properties can be marked "const", which are equivalent to frozen data properties in Javascript. Function-valued properties are considered "methods" of a trait. A required property "s : required" is equivalent to a property bound to `traits.js`'s `Trait.required` marker.

Syntax

$$\begin{array}{rcl}
e \in \textbf{Expr} & ::= & \text{this} \mid x \mid \text{null} \mid e \,;\, e \mid \text{let}\, x = e \,\text{in}\, e \mid e.s \mid e.s = e \\
 & \mid & e.s(\overline{e}) \mid \text{trait}\{\overline{p}\} \mid \text{newTrait}\, e \mid \text{newObject}\, e \\
 & \mid & \text{compose}\, e\, e \mid \text{override}\, e\, e \mid \text{resolve}\, \overline{s \mapsto a}\, e \\
p \in \textbf{PropDecl} & ::= & s : e \mid \text{const}\, s : e \mid s : \text{function}(\overline{x})\{e\} \mid s : \text{required} \\
a \in \textbf{Alias} & ::= & s \mid \text{null}
\end{array}$$

Syntactic Sugar

$$e \,;\, e' \overset{\text{def}}{=} \text{let}\, x = e \,\text{in}\, e' \quad x \notin \text{FV}(e')$$

Evaluation Contexts and Runtime Syntax

$$\begin{array}{rcl}
e_\Box & ::= & \Box \mid \text{let}\, x = e_\Box \,\text{in}\, e \mid e_\Box.s \mid e_\Box.s = e \mid v.s = e_\Box \mid e_\Box.s(\overline{e}) \mid v.s(\overline{v}, e_\Box, \overline{e}) \\
 & \mid & \text{trait}\{\overline{p}\, p_\Box\, \overline{p}\} \mid \text{newTrait}\, e_\Box \mid \text{newObject}\, e_\Box \mid \text{compose}\, e_\Box\, e \\
 & \mid & \text{compose}\, v\, e_\Box \mid \text{override}\, e_\Box\, e \mid \text{override}\, v\, e_\Box \mid \text{resolve}\, \overline{s \mapsto a}\, e_\Box \\
p_\Box & ::= & s : e_\Box \mid \text{const}\, s : e_\Box \\
 & & \\
e & ::= & \ldots \mid v
\end{array}$$

In TRAITS-JS, objects can only be created by instantiating traits. The expression `newTrait t` instantiates a trait as if by `Trait.create(t)`. The expression `newObject t` instantiates a trait as if by `Object.create(t)`.

The `compose`, `override` and `resolve` operators can be used to define composite traits. They model the equivalent functions provided by `traits.js`. The `resolve` operator takes as its first argument a sequence of aliases, which map strings to either new strings to denote renaming, or to the `null` value to denote exclusion, as in `traits.js`.

Finally, expression sequencing $e; e'$ is simply syntactic sugar for a `let`-expression that binds the result of e to a variable that is not free in e'.

Evaluation Contexts and Runtime Expressions. We use evaluation contexts [14] to indicate what subexpressions of an expression should be fully reduced before the compound expression itself can be further reduced. $e_\Box$ denotes an expression with a "hole". Each appearance of $e_\Box$ indicates a subexpression with a possible hole. The intent is for the hole to identify the next subexpression to reduce in a compound expression.

Our reduction rules operate on "runtime expressions", which are simply all expressions including values v, as a subexpression may reduce to a value before being reduced further.

Substitution Rules

$$\begin{array}{rcl}
[v/x]x & = & v \\
[v/x]x' & = & x' \\
[v/x]v' & = & v' \\
[v/x]e.s & = & ([v/x]e).s \\
[v/x]e.s(\overline{e}) & = & [v/x]e.s([v/x]\overline{e})
\end{array}
\qquad
\begin{array}{rcl}
[v/x]e.s = e & = & ([v/x]e).s = [v/x]e \\
[v/x]s\ :\ e & = & s\ :\ [v/x]e \\
[v/x]\text{const}\ s : e & = & \text{const}\ s : [v/x]e \\
[v/x]s : \text{required} & = & s : \text{required}
\end{array}$$

$$\begin{array}{rcll}
[v/x]s : \text{function}(\overline{x})\{e\} & = & s : \text{function}(\overline{x})\{e\} & \text{if } x \in \overline{x} \\
[v/x]s : \text{function}(\overline{x})\{e\} & = & s : \text{function}(\overline{x})\{[v/x]e\} & \text{if } x \notin \overline{x} \\
[v/x]\text{let}\ x' = e \text{ in } e & = & \text{let}\ x' = [v/x]e \text{ in } [v/x]e & \\
[v/x]\text{let}\ x = e \text{ in } e & = & \text{let}\ x = [v/x]e \text{ in } e & \\
[v/x]\text{compose}\ e\ e' & = & \text{compose}\ [v/x]e\ [v/x]e' & \\
[v/x]\text{override}\ e\ e' & = & \text{override}\ [v/x]e\ [v/x]e' & \\
[v/x]\text{newTrait}\ e & = & \text{newTrait}\ [v/x]e & \\
[v/x]\text{newObject}\ e & = & \text{newObject}\ [v/x]e & \\
[v/x]\text{resolve}\ \overline{s \mapsto a}\ e & = & \text{resolve}\ \overline{s \mapsto a}\ [v/x]e & \\
[v/x]\text{trait}\{\overline{p}\} & = & \text{trait}\{\overline{[v/x]p}\} & \text{if } x \neq \text{this} \\
[v/\text{this}]\text{trait}\{\overline{p}\} & = & \text{trait}\{\overline{p}\} &
\end{array}$$

Figure 3: Substitution rules: x denotes a variable name or the pseudovariable this.

7.2. Reduction Rules

The reduction rules describe property lookup, property update, trait creation, trait composition and trait instantiation in TRAITS-JS. Before explaining each rule in detail, we explain some notational conventions.

Notation. Our notation is mostly derived from the operational semantics of JCoBox [15]. $\overline{v}$ denotes a sequence of items, with ϵ denoting the empty sequence. The notation $v \cdot \overline{v}$ deconstructs a sequence into a subsequence $\overline{v}$ and its first element v. We use the notation $S' = S \cup \{s\}$ to lookup and extract an element s from the set S, such that $s \in S', S = S' \setminus \{s\}$.

The function $\text{dom}(f)$ denotes the domain of a partial function f as a set. The notation $f[s \mapsto v]$ denotes a function that extends the domain of f with a new element s. If $s \in \text{dom}(f)$, the extended function overrides the previous value of $f(s)$.

The notation $e_\Box[e]$ indicates that the expression e is (potentially) part of a compound expression $e_\Box$, and should be reduced first before the compound expression can be reduced further.

Variables. Variables are introduced via `let`-expressions. The rule [LET] describes that such an expression reduces to the body e, with v substituted for x in e, where the heap H remains unchanged. The precise semantics of variable substitution is given in Figure 3.

Trait construction. The rule [CONSTRUCT-TRAIT] describes that an expression "trait $\{\overline{p}\}$" reduces to an object reference ι_o, where ι_o is a fresh identity not yet present in the heap H. After executing this rule, the heap is updated with a reference to the new trait t.

The auxiliary function "construct" (see p.19) describes t's partial function f from strings s to properties p, given the property declarations $\overline{p}$. The function $f_\emptyset$ denotes a function whose domain is the empty set $\emptyset$, i.e. $f_\emptyset$ is undefined for any string s.

For any data property declaration $s : v$ in $\overline{p}$, t's partial function f maps s to a mutable data property $\mathcal{D}\langle v, \text{true}\rangle$. Similarly, for any constant data property declaration const $s : v$, $f(s) = \mathcal{D}\langle v, \text{false}\rangle$. Function property declarations $s :$ function$(\overline{x})\{e\}$ are mapped to method properties $\mathcal{M}\langle \overline{x}, e\rangle$. Finally, required property declarations "s : required" are mapped to required properties $\mathcal{R}$.

Property access and update. The rule [ACCESS-DATA-PROPERTY] describes that accessing a property s on an object reference ι_o reduces to a value v, if ι_o refers to a valid object in the heap H and that object's partial function f maps s to a data property $\mathcal{D}\langle v, b'\rangle$. The mutability b' of the data property is irrelevant.

The rule [ACCESS-MISSING-PROPERTY] describes property lookup when the property s is not present in the target object ι_o. Such property lookups always reduce to `null`.

(let)

$$H, e_\Box[\text{let } x = v \text{ in } e] \rightarrow H, e_\Box[[v/x]e]$$

(construct-trait)

$$\frac{\iota_o \notin \text{dom}(H) \qquad t = \mathcal{T}\langle f\rangle \qquad f = \text{construct}(\overline{p}, f_\emptyset)}{H, e_\Box[\text{trait}\{\overline{p}\}] \rightarrow H[\iota_o \mapsto t], e_\Box[\iota_o]}$$

(access-data-property)

$$\frac{H(\iota_o) = \mathcal{O}\langle f, b\rangle \qquad f(s) = \mathcal{D}\langle v, b'\rangle}{H, e_\Box[\iota_o.s] \rightarrow H, e_\Box[v]}$$

(access-missing-property)

$$\frac{H(\iota_o) = \mathcal{O}\langle f, b\rangle \qquad s \notin \text{dom}(f)}{H, e_\Box[\iota_o.s] \rightarrow H, e_\Box[\text{null}]}$$

(invoke-method)

$$\frac{H(\iota_o) = \mathcal{O}\langle f, b\rangle \qquad f(s) = \mathcal{M}\langle \overline{x}, e\rangle}{H, e_\Box[\iota_o.s(\overline{v})] \rightarrow H, e_\Box[[\iota_o/\text{this}][\overline{v}/\overline{x}]e]}$$

(invoke-unbound-function)

$$\frac{\begin{array}{c}H(\iota_o) = \mathcal{O}\langle f, b\rangle \qquad f(s) = \mathcal{D}\langle \iota_{o'}, b\rangle \\ H(\iota_{o'}) = \mathcal{F}\langle \overline{x}, e, \text{null}\rangle\end{array}}{H, e_\Box[\iota_o.s(\overline{v})] \rightarrow H, e_\Box[[\iota_o/\text{this}][\overline{v}/\overline{x}]e]}$$

(invoke-bound-function)

$$\frac{\begin{array}{c}H(\iota_o) = \mathcal{O}\langle f, b\rangle \qquad f(s) = \mathcal{D}\langle \iota_{o'}, b\rangle \\ H(\iota_{o'}) = \mathcal{F}\langle \overline{x}, e, \iota_{\text{this}}\rangle\end{array}}{H, e_\Box[\iota_o.s(\overline{v})] \rightarrow H, e_\Box[[\iota_{\text{this}}/\text{this}][\overline{v}/\overline{x}]e]}$$

(access-method-property)

$$\frac{\begin{array}{c}H(\iota_o) = \mathcal{O}\langle f, b\rangle \qquad f(s) = \mathcal{M}\langle \overline{x}, e\rangle \\ \iota_{o'} \notin \text{dom}(H) \qquad o = \mathcal{F}\langle \overline{x}, e, \text{null}\rangle\end{array}}{H, e_\Box[\iota_o.s] \rightarrow H[\iota_{o'} \mapsto o], e_\Box[\iota_{o'}]}$$

(update-data-property)

$$\frac{\begin{array}{c}H(\iota_o) = \mathcal{O}\langle f, b\rangle \qquad f(s) = \mathcal{D}\langle v', \text{true}\rangle \\ o' = \mathcal{O}\langle f[s \mapsto \mathcal{D}\langle v, \text{true}\rangle], b\rangle\end{array}}{H, e_\Box[\iota_o.s = v] \rightarrow H[\iota_o \mapsto o'], e_\Box[v]}$$

(update-missing-property)

$$\frac{\begin{array}{c}H(\iota_o) = \mathcal{O}\langle f, \text{true}\rangle \qquad s \notin \text{dom}(f) \\ o' = \mathcal{O}\langle f[s \mapsto \mathcal{D}\langle v, \text{true}\rangle], \text{true}\rangle\end{array}}{H, e_\Box[\iota_o.s = v] \rightarrow H[\iota_o \mapsto o'], e_\Box[v]}$$

(compose)

$$\frac{\begin{array}{c}H(\iota_{o_1}) = \mathcal{T}\langle f_1\rangle \quad H(\iota_{o_2}) = \mathcal{T}\langle f_2\rangle \quad t = \mathcal{T}\langle f\rangle \\ \iota_o \notin \text{dom}(H) \qquad f = \text{compose}(\text{dom}(f_1), f_1, f_2)\end{array}}{H, e_\Box[\text{compose } \iota_{o_1}\ \iota_{o_2}] \rightarrow H[\iota_o \mapsto t], e_\Box[\iota_o]}$$

(override)

$$\frac{\begin{array}{c}H(\iota_{o_1}) = \mathcal{T}\langle f_1\rangle \quad H(\iota_{o_2}) = \mathcal{T}\langle f_2\rangle \quad t = \mathcal{T}\langle f\rangle \\ \iota_o \notin \text{dom}(H) \qquad f = \text{override}(f_1, f_2)\end{array}}{H, e_\Box[\text{override } \iota_{o_1}\ \iota_{o_2}] \rightarrow H[\iota_o \mapsto t], e_\Box[\iota_o]}$$

(resolve)

$$\frac{\begin{array}{c}H(\iota_o) = \mathcal{T}\langle f\rangle \qquad \iota_{o'} \notin \text{dom}(H) \\ t = \mathcal{T}\langle f'\rangle \qquad f' = \text{resolve}(\overline{s \mapsto a}, f)\end{array}}{H, e_\Box[\text{resolve } \overline{s \mapsto a}\ \iota_o] \rightarrow H[\iota_{o'} \mapsto t], e_\Box[\iota_{o'}]}$$

(new-object)

$$\frac{\begin{array}{c}H(\iota_o) = \mathcal{T}\langle f\rangle \qquad \iota_{o'} \notin \text{dom}(H) \\ o = \mathcal{O}\langle f', \text{true}\rangle \qquad f' = \text{instantiate}(f)\end{array}}{H, e_\Box[\text{newObject } \iota_o] \rightarrow H[\iota_{o'} \mapsto o], e_\Box[\iota_{o'}]}$$

(new-trait)

$$\frac{\begin{array}{c}H(\iota_o) = \mathcal{T}\langle f\rangle \quad \iota_{o'} \notin \text{dom}(H) \quad o = \mathcal{O}\langle f', \text{false}\rangle \\ \nexists s \in \text{dom}(f) : f(s) = \mathcal{R} \vee f(s) = \mathcal{C} \\ H', f' = \text{freeze}(\text{dom}(f), f, \iota_{o'}, H, f_\emptyset)\end{array}}{H, e_\Box[\text{newTrait } \iota_o] \rightarrow H'[\iota_{o'} \mapsto o], e_\Box[\iota_{o'}]}$$

Auxiliary functions

$$
\begin{aligned}
\mathrm{construct}(\epsilon, f) &\overset{def}{=} f \\
\mathrm{construct}((s\ :\ v)\cdot \overline{p}, f) &\overset{def}{=} \mathrm{construct}(\overline{p}, f[s \mapsto \mathcal{D}\langle v, \mathrm{true}\rangle]) \\
\mathrm{construct}((\mathrm{const}\ s : v)\cdot \overline{p}, f) &\overset{def}{=} \mathrm{construct}(\overline{p}, f[s \mapsto \mathcal{D}\langle v, \mathrm{false}\rangle]) \\
\mathrm{construct}((s : \mathrm{function}(\overline{x})\{e\})\cdot \overline{p}, f) &\overset{def}{=} \mathrm{construct}(\overline{p}, f[s \mapsto \mathcal{M}\langle \overline{x}, e\rangle]) \\
\mathrm{construct}((s : \mathrm{required})\cdot \overline{p}, f) &\overset{def}{=} \mathrm{construct}(\overline{p}, f[s \mapsto \mathcal{R}])
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{compose}(\emptyset, f_1, f_2) &\overset{def}{=} f_2 \\
\mathrm{compose}(\{s\}\cup S, f_1, f_2) &\overset{def}{=} \begin{cases} \mathrm{compose}(S, f_1, f_2[s \mapsto f_1(s)]) & \text{if } s \notin \mathrm{dom}(f_2) \\ \mathrm{compose}(S, f_1, f_2[s \mapsto f_1(s) \oplus f_2(s)]) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\mathrm{override}(f_1, f_2) \overset{def}{=} f(s) = \begin{cases} f_1(s) & \text{if } s \in \mathrm{dom}(f_1) \wedge f_1(s) \neq \mathcal{R} \\ \mathcal{R} & \text{if } s \notin \mathrm{dom}(f_2) \wedge f_1(s) = \mathcal{R} \\ f_2(s) & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
\mathrm{resolve}(\epsilon, f) &\overset{def}{=} f \\
\mathrm{resolve}(s \mapsto s' \cdot \overline{a}, f) &\overset{def}{=} \begin{cases} \mathrm{resolve}(\overline{a}, f) & \text{if } s \notin \mathrm{dom}(f) \\ \mathrm{resolve}(\overline{a}, f[s \mapsto \mathcal{R}, s' \mapsto f(s) \oplus f(s')]) & \text{if } s, s' \in \mathrm{dom}(f) \\ \mathrm{resolve}(\overline{a}, f[s \mapsto \mathcal{R}, s' \mapsto f(s)]) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\mathrm{resolve}(s \mapsto \mathrm{null} \cdot \overline{a}, f) \overset{def}{=} \begin{cases} \mathrm{resolve}(\overline{a}, f) & \text{if } s \notin \mathrm{dom}(f) \\ \mathrm{resolve}(\overline{a}, f[s \mapsto \mathcal{R}]) & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
\mathrm{freeze}(\emptyset, f, \iota_o, H, f') &\overset{def}{=} H, f' \\
\mathrm{freeze}(\{s\}\cup S, f, \iota_o, H, f') &\overset{def}{=} \begin{cases} \mathrm{freeze}(S, f, \iota_o, H, f'[s \mapsto \mathcal{D}\langle v, \mathrm{false}\rangle]) \\ \qquad \text{if } f(s) = \mathcal{D}\langle v, b\rangle \\ \mathrm{freeze}(S, f, \iota_o, H[\iota_{o'} \mapsto \mathcal{F}\langle \overline{x}, e, \iota_o\rangle], f'[s \mapsto \mathcal{D}\langle \iota_{o'}, \mathrm{false}\rangle]) \\ \qquad \text{if } f(s) = \mathcal{M}\langle \overline{x}, e\rangle \text{ where } \iota_{o'} \notin \mathrm{dom}(H) \end{cases}
\end{aligned}
$$

$$
\mathrm{instantiate}(f) \overset{def}{=} f'(s) = \begin{cases} \mathcal{D}\langle \mathrm{null}, \mathrm{true}\rangle & \text{if } f(s) = \mathcal{R} \\ f(s) & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
\mathcal{D}\langle v, b\rangle \oplus \mathcal{M}\langle \overline{x}, e\rangle &\overset{def}{=} \mathcal{C} \\
\mathcal{D}\langle v_1, b_1\rangle \oplus \mathcal{D}\langle v_2, b_2\rangle &\overset{def}{=} \mathcal{C} \\
\mathcal{M}\langle \overline{x}, e\rangle \oplus \mathcal{M}\langle \overline{x'}, e'\rangle &\overset{def}{=} \mathcal{C} \\
p \oplus \mathcal{C} &\overset{def}{=} \mathcal{C} \\
p \oplus \mathcal{R} &\overset{def}{=} p \\
p \oplus p &\overset{def}{=} p \\
p_1 \oplus p_2 &\overset{def}{=} p_2 \oplus p_1
\end{aligned}
$$

The rule [INVOKE-METHOD] states that invoking a method property amounts to evaluating the method body e with proper substitution of actual arguments $\overline{v}$ for formal parameters $\overline{x}$, and with the `this` pseudovariable replaced by the receiver ι_o.

The rule [INVOKE-UNBOUND-FUNCTION] is similar to [INVOKE-METHOD], except that this time, s refers to a function-valued data property. The function-valued property has a `null`-valued `this`-binding, so that the `this` pseudovariable in the invoked function body will be bound to the receiver of the method invocation ι_o.

The rule [INVOKE-BOUND-FUNCTION] differs from [INVOKE-UNBOUND-FUNCTION] only in the treatment of the `this`-binding: instead of using the receiver ι_o, the `this`-binding ι_{this} stored within the function is used.

The rule [ACCESS-METHOD-PROPERTY] describes that when accessing a method property, the property is extracted from the object as a fresh first-class function. This new function o is unbound (i.e. it has a `null`-valued `this`-binding). In TRAITS-JS, as in Javascript, methods are not automatically bound on extraction.

The rule [UPDATE-DATA-PROPERTY] describes an update to a *mutable* data property $\mathcal{D}\langle v', \text{true}\rangle$. No reduction rule is applicable to update an immutable data property.

The rule [UPDATE-MISSING-PROPERTY] describes an update to a non-existent property s. If the receiver object ι_o is extensible, the property s is added to the object. No reduction rule is applicable to extend a non-extensible receiver object with new properties.

Trait composition. The rule [COMPOSE] is applicable only if its two argument values ι_{o_1} and ι_{o_2} both refer to existing trait values. The partial functions f_1 and f_2 of these traits are combined into a composite function f. This composition is described in terms of structural induction on the domain of f_1. Operationally, each string s in the domain of f_1 is combined with f_2. If s does not appear in f_2, then $f(s) = f_1(s)$. If s appears in the domain of both f_1 and f_2, the properties are combined using the $\oplus$ operator.

The $\oplus$ operator is commutative and associative, with a zero element $\mathcal{C}$ and a neutral element $\mathcal{R}$. If the combined properties p_1 and p_2 are data or method properties, $p_1 \oplus p_2$ is always a conflicting property unless the properties are identical.

The rule [OVERRIDE] is similar to [COMPOSE]. Here, f_1 and f_2 are combined such that any properties defined on f_1 always take precedence over any properties defined on f_2. Required properties form an exception: if $f_1(s) = \mathcal{R}$ and $f_2(s)$ is defined, then $f_2(s)$ overrides the required property of f_1. It is easy to see that if f_1 and f_2 contain no conflicting properties, then override(f_1, f_2) will also never contain conflicting properties.

The rule [RESOLVE] describes renaming and exclusion of properties from an existing trait. The auxiliary function "resolve" processes the aliases in sequence. An alias of the form $s \mapsto s'$ denotes a renaming. If s does not exist as the property of the trait, the renaming has no effect. Otherwise, in the resolved trait, s becomes a required property $\mathcal{R}$. If both s and s' exist as properties of the trait, in the resolved trait, s' is bound to the composition of s and s', otherwise s' refers to whatever property s referred to in the original trait.

An alias of the form $s \mapsto \text{null}$ denotes an exclusion. The property s becomes a required property $\mathcal{R}$ in the resolved trait. As with renaming, excluding a non-existent property has no effect on the resolved trait.

Trait instantiation. The rule [NEW-OBJECT] describes that the "newObject" operator always evaluates to a fresh, extensible object. The object's partial function is derived from that of its trait. The only difference between f' and f is that f' transforms all required properties $\mathcal{R}$ in f into data properties $\mathcal{D}\langle\text{null}, \text{true}\rangle$. In other words, required properties are represented on objects as normal data properties bound to `null`.

Note that the function f' may still contain conflicting properties $\mathcal{C}$. This does not prevent the newly created object from being used, but since there are no reduction rules applicable for looking up, assigning to or invoking a conflicting property, a program will get stuck when trying to manipulate a conflicting property. This reflects the behavior that in `traits.js`, accessing or updating a conflicting property throws an exception.

The rule [NEW-TRAIT] describes trait instantiation. A first important difference with [NEW-OBJECT] is that this rule is only applicable if the trait's function f does not contain any required or conflicting properties. This prevents incomplete or inconsistent traits from being instantiated using "newTrait".

The second difference is that the new instance's partial function f' is derived from f such that all data properties of f are frozen in f'. Moreover, all method properties of f are turned into *bound* function-valued data properties on f'. This guarantees that these properties can only be extracted as bound functions, such that clients will never be able to modify the `this`-binding within these methods. Allocating a bound function per method property requires modifying the heap, which is why the "freeze" function describes both the updated function f' as well as the updated heap H' which contains the new trait's bound functions.

A third difference is that [NEW-TRAIT] defines the new instance o as a non-extensible object, while [NEW-OBJECT] defines o as an extensible object.

7.3. Example

As mentioned previously, one of the reasons for formalizing `traits.js` is to gain a better insight into the precise rules of trait composition and renaming. Armed with the calculus, we can for instance answer questions such as what is the semantics when two properties are renamed to the same alias. To answer this question, we can derive the structure of the following trait definition:

```
Trait.resolve({ a: 'c', b: 'c' }, Trait({ a: 1, b: 2 }))
```

Or, restated in TRAITS-JS (assuming numbers are legal values):

$$\text{resolve}\ (a \mapsto c, b \mapsto c)\ (\text{trait}\{a : 1, b : 2\})$$

Reducing the "resolve" operator according to the [RESOLVE] rule involves applying the auxiliary "resolve" function defined on p. 19, as follows:

$$\begin{array}{lll} & \text{resolve}(\{a \mapsto c, b \mapsto c\}, f) & \text{where } f = f_{\emptyset}[a \mapsto \mathcal{D}\langle 1, \text{true}\rangle, b \mapsto \mathcal{D}\langle 2, \text{true}\rangle] \\ = & \text{resolve}(\{b \mapsto c\}, f') & \text{where } f' = f[a \mapsto \mathcal{R}, c \mapsto \mathcal{D}\langle 1, \text{true}\rangle] \\ = & \text{resolve}(\{\}, f'') & \text{where } f'' = f'[b \mapsto \mathcal{R}, c \mapsto \mathcal{D}\langle 1, \text{true}\rangle \oplus \mathcal{D}\langle 2, \text{true}\rangle] \\ = & f_{\emptyset}[a \mapsto \mathcal{R}, b \mapsto \mathcal{R}, c \mapsto \mathcal{C}] & \end{array}$$

In other words, we can verify that renaming two properties to the same property leads to a conflict.

7.4. Related Work

Trait-based object composition has previously been formalized. Bergel *et. al* [16] have formalized trait composition with support for stateful traits in the SMALLTALKLITE calculus. They similarly formalize trait composition, overriding, aliasing and exclusion. Contrary to TRAITS-JS, SMALLTALKLITE is class-based, has no notion of constant properties, non-extensible objects or methods that can be extracted as first-class functions. TRAITS-JS also does not treat state (data properties) differently from method properties.

Formal work on traits in Java-like languages (i.e. statically typed and class-based) include Smith and Drossopoulou [11]'s Chai language, Liquori and Spiwack's FeatherTrait Java [12] and Reppy and Turon's Meta-trait Java [13]. In these formal models, traits are typically not first-class values, and objects in Java-like languages are high-integrity by default: they are non-extensible and their methods cannot be extracted as first-class unbound function values.

The purpose of TRAITS-JS is not to accurately formalize Javascript. For a more accurate formal semantics of Javascript, see [17].

7.5. Summary

While the TRAITS-JS calculus models only a small subset of Javascript, it provides a fairly complete coverage of the `traits.js` library semantics. In particular, it makes explicit the `this`-binding of methods in the presence of method extraction (rule [ACCESS-METHOD-PROPERTY]) and method binding (rule [INVOKE-BOUND-FUNCTION]). Moreover, it shows how `Trait.create` generates tamper-proof instances (rule [NEW-TRAIT]), as opposed to `Object.create`, which generates extensible objects with potentially mutable properties (rule [NEW-OBJECT]). Finally, the rules for trait composition and renaming are detailed, making more explicit the fact that trait composition is indeed commutative and associative.

8. Discussion

8.1. Traits and Inheritance

As noted in Section 3, traits were originally defined as reusable groups of methods that can be composed together to form a class [5]. Classes can further be composed using standard inheritance, which is still useful to e.g. inherit instance variables from superclasses (since traits as originally proposed were stateless).

In `traits.js`, traits are not composed into classes (or constructor functions, which is the closest analog to a class in Javascript). Instead, the philosophy of `traits.js` is to compose atomic traits into larger, composite traits, and then to *directly* instantiate those traits into instances (i.e. objects), without any intermediate class-like abstraction. Since traits can be stateful, there is no need for a separate class-hierarchy to introduce state.

While Javascript has no notion of class-based inheritance, it does feature prototypal inheritance, which is the language's primitive object composition mechanism. The question remains how trait instances interact with this prototypal inheritance. That is: can a trait *instance* still serve as a *prototype* object so that it can be further extended using prototypal inheritance?

The answer depends on how the trait instance was instantiated. Trait instances created using `Trait.create` do not compose well with prototypal inheritance, as the following example illustrates:

```
var instance = Trait.create(Object.prototype, Trait({
  message: 'hello world',
  greeting: function() { return this.message; }
}));
var child = Object.create(instance);
child.message = 'goodbye world';
child.greeting(); // returns 'hello world'
```

In this example, `instance` is a trait instance used as the prototype of another `child` object. The `child` object inherits the `instance`'s `greeting` method, but overrides the `message` property. However, when invoking the `greeting` method, the method fails to return the overridden property. This is because trait instances generated using `Trait.create` have only *bound* methods: the `this`-pseudovariable inside the `greeting` method is bound to the `instance` object.

As stated in Section 6, the explicit goal of `Trait.create` was to generate high-integrity instances. To guarantee high-integrity, the instance should be a self-contained unit. It is no longer meant to be a unit of reuse. Clients of such instances can only invoke the instance's methods and read its properties. Being able to inherit and override methods from such instances is explicitly not part of the contract. In a nutshell: high-integrity objects don't compose (but the traits from which they were instantiated do).

Trait instances generated using `Object.create` are normal Javascript objects without any restrictions to support high-integrity. As a result, such objects can be freely used as prototype objects and their methods may be overridden by child objects.

8.2. 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.

9. Validation

9.1. 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 i7 2.4Ghz Macbook Pro with 8GB of memory, running Mac OS X 10.7.3 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 roughly a factor of 10 slower than creating

	Firefox 11.0		**Chrome 17.0.963.79**		**Safari 5.1.3 (7534.53.10)**	
alloc.	Trait.create	Object.create	Trait.create	Object.create	Trait.create	Object.create
size 10	8.56x ±.62	1.04x ±.08	9.00x ±.59	.71x ±.04	3.98x ±1.33	.48x ±.18
size 100	9.45x±.27	1.00x ±.01	11.42x ±.17	.98x ±.01	7.70x ±.25	1.11x ±.04
size 1000	7.85x±.09	.91x ±.01	11.20x ±.06	.97x ±.01	6.65x ±.24	1.04x ±.01
meth call	5.53x ±.93	.93x ±.13	15.30x ±3.40	1.30x ±.60	2.28x ±.36	1.02x ±.19

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

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 (> 1.0 indicates a slowdown, < 1.0 a speedup). Each number is the mean ratio of 5 runs (each in an independent, warmed-up browser session, performing the operation 1000 times), ± 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`.

Across different platforms and sizes, there is on average a factor of 8.42x 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` compared to instantiating regular property descriptors. But to repeat, `Object.create` is itself roughly 10 times 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 2.28 to 15.30x slowdown compared to a standard method invocation. The large differences among platforms stem from the dependence on the implementation of bound methods, which are fairly rare in regular Javascript code, and thus far less optimized than regular methods calls. Again, for instances created by `Object.create` there is no overhead, since such instances do not have bound methods.

These micro-benchmarks provide little insight into the overhead of `traits.js` when used in realistic Javascript applications. This type of overhead is studied in the next Section.

9.2. Morphic UI Framework

`traits.js` has been used to build a small UI widget library in the browser. The widgets are rendered using an HTML5 Canvas, a standard API to perform 2D graphics in the browser. The UI widget library was inspired by Morphic, the UI framework of the Self programming language [18]. The library comprises roughly 2800 lines of Javascript code and defines 18 traits. The core widget, named `BaseMorph`, is a trait that is itself composed out of 5 composite traits, to reuse behavior related to the calculation of bounding boxes, collections (composite morphs are represented as hierarchical trees), coloring and animation.

Figure 4 shows a screenshot of the Morphic bouncing atoms demo, built using `traits.js` and running in a browser. The demo renders animated atoms (circles) that bounce around within a gas (the green area). The slider on the right is used to control temperature (influencing the speed of the atoms). With 10 bouncing atoms, the demo achieves a smooth 40 frames per second (each visual widget on the screen is a trait instance). Figure 5 shows the same browser session where the user re-arranged the widgets, demonstrating that each widget on the screen is a malleable object, according to the philosophy of Morphic.

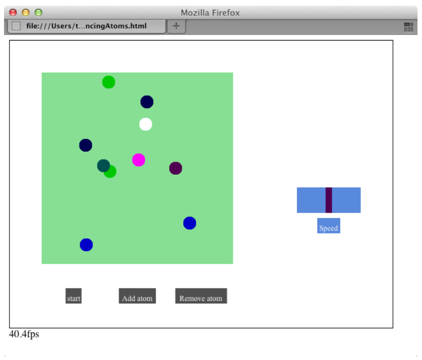

Figure 4: Bouncing atoms demo in initial configuration (animating at 40fps).

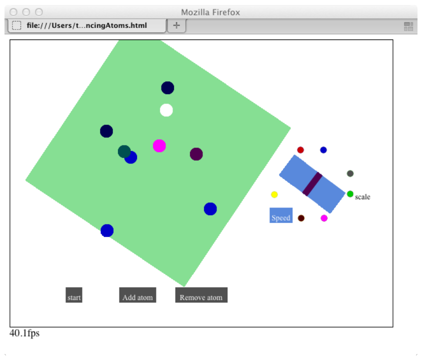

Figure 5: Bouncing atoms demo with displaced morphs.

Naturally, a UI widget library exposes lots of opportunity for reuse and for the creation of extensive abstraction hierarchies. However, the library does not make use of `traits.js`'s support for creating high-integrity objects: all trait instances are created using `Object.create`. To test the cost of `Trait.create` versus `Object.create` in a more realistic setting, we changed the code so all trait instances would be created using `Trait.create`. Since the trait instances in our application defined mutable state (e.g. the position of a morph), we refactored all such mutable state into accessor properties that modify a lexically enclosing variable. The measured overhead was acceptable: over a 60 second time window, our modified demo achieved an average 39.42fps, compared to 40.96fps for the original (a slowdown of 3.76%).

10. Conclusion

`traits.js` is a small, standards-compliant trait composition library for Javascript. Compared to the object composition functionality of most popular Javascript libraries, it provides support for true trait-based (as opposed to mixin-based) composition. 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 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. In web content where mutually distrusting scripts have to cooperate, this ability to conveniently define high-integrity objects is a useful addition to the Javascript programmer's toolbox.

Acknowledgements

We thank the members of the ECMAScript committee and the `es-discuss` mailing list for their valuable feedback. We also thank the anonymous reviewers for their constructive feedback on an earlier version of this paper.

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

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

[3] T. Van Cutsem, M. S. Miller, Traits.js: robust object composition and high-integrity objects for ECMAScript 5, in: Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients, PLASTIC '11, ACM, New York, NY, USA, 2011, pp. 1–8.

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

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

[6] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. P. Black, Traits: A mechanism for fine-grained reuse, ACM Trans. Program. Lang. Syst. 28 (2006) 331–388.

[7] G. Bracha, W. Cook, Mixin-based inheritance, in: OOPSLA/ECOOP '90, ACM, New York, NY, USA, 1990, pp. 303–311.

[8] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, in: OOPSLA '86, ACM, New York, NY, USA, 1986, pp. 38–45.

[9] M. Flatt, R. B. Finder, M. Felleisen, Scheme with classes, mixins and traits, in: AAPLAS '06.

[10] K. Fisher, J. Reppy, Statically typed traits, Technical Report TR-2003-13, University of Chicago, Department of Computer Science, 2003.

[11] C. Smith, S. Drossopoulou, Chai: Typed traits in Java, in: Proceedings ECOOP 2005.

[12] L. Liquori, A. Spiwack, FeatherTrait: A modest extension of Featherweight Java, ACM Transactions on Programming Languages and Systems (TOPLAS) 30 (2008) 1–32.

[13] J. Reppy, A. Turon, Metaprogramming with traits, in: Proceedings of European Conference on Object-Oriented Programming (ECOOP'2007).

[14] M. Felleisen, R. Hieb, The revised report on the syntactic theories of sequential control and state, Theor. Comput. Sci. 103 (1992) 235–271.

[15] J. Schäfer, A. Poetzsch-Heffter, JCoBox: generalizing active objects to concurrent components, in: Proceedings of the 24th European conference on Object-oriented programming, ECOOP'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 275–299.

[16] A. Bergel, S. Ducasse, O. Nierstrasz, R. Wuyts, Stateful traits and their formalization, Journal of Computer Languages, Systems and Structures 34 (2007) 83–108.

[17] A. Guha, C. Saftoiu, S. Krishnamurthi, The essence of Javascript, in: Proceedings of the 24th European Conference on Object-oriented Programming, ECOOP'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 126–150.

[18] J. H. Maloney, R. B. Smith, Directness and liveness in the morphic user interface construction environment, in: Proceedings of the 8th annual ACM symposium on User interface and software technology, UIST '95, ACM, New York, NY, USA, 1995, pp. 21–28.