[[strawman: syntax_for_efficient_traits]]

ES Wiki

Syntax for Efficient Traits

The Traits library deals in individual trait instances. Without VM support, this is unpleasantly expensive, requiring at least an allocation per method per instance. On the one hand, the full expressiveness of the Traits library requires it to operate dynamically on trait instances. On the other hand, the expected typical usage pattern wraps trait operations in functions for making traits, where an individual trait-making-function would typically make traits with the same method code coupled with different data. Given adequate analysis, this more static pattern should be easily optimizable by VMs into a vtable-based implementation, where the method code gathered together by a class is painlessly shared among its instances.

The syntax for traits presented here supports this more static expected pattern of use, where a *trait class* serves as a trait-making-function. This serves several purposes:

- It provides a more convenient and understandable notation for this pattern.
- It avoids the need for implementers to recognize and optimize a complex pattern of using the Traits library. Instead, implementers need only optimize uses of this notation.
- It helps programmers know when to expect an efficient implementation. Traits expressed using this notation are either erroneous and will fail early (on evaluating a ClassDeclaration or ClassExpression at the latest) or will be efficiently implemented.

None of this precludes the generality supported by the Traits library. The Traits library and Object.create remain available and interoperate smoothly with code expressed using this syntax, without creating confusion for either programmers or implementers about what is supposed to be efficient.

```
Declaration : ...           // "..." means all the existing members
    LetDeclaration
    ConstDeclaration
    FunctionDeclaration
    ClassDeclaration
    TraitClassDeclaration
ClassDeclaration :          // by analogy with FunctionDeclaration
    class Identifier ( FormalParameterList? ) TraitBody
TraitClassDeclaration :     // by analogy with FunctionDeclaration
    trait class Identifier ( FormalParameterList? ) TraitBody
MemberExpression : ...
    ClassExpression
    TraitClassExpression
ClassExpression :           // by analogy with FunctionExpression
    class Identifier? ( FormalParameterList? ) TraitBody
TraitClassExpression :      // by analogy with FunctionExpression
    trait class Identifier? ( FormalParameterList? ) TraitBody
```

```
TraitBody :                 // by analogy with FunctionBody
    => TraitLiteral
    { Program => TraitLiteral }
```

A *class* makes *instances* that behave according to the behavior described by the class. A *trait class* makes *traits*, which can contribute towards a class' description of the behavior of its instances. TraitBody is defined using "=>" and a second production so that a class or trait class could initialize private state before constructing the instance or trait that might use this state.

By "Program" above, we simply mean an optional list of SourceElements excluding ReturnStatements. But for this exclusion, what we mean by "Program" is identical to FunctionBody.

The TraitLiteral

```
TraitLiteral :   // by analogy to ObjectLiteral
    { TraitMixinList? , TraitPartList? } // ignoring obvious comma placement issues
TraitMixinList : // by analogy to PropertyNameAndValueList
    TraitMixin
    TraitMixinList , TraitMixin
TraitPartList :  // by analogy to PropertyNameAndValueList
    TraitPart
    TraitPartList , TraitPart

TraitMixin :
    mixin Identifier Arguments Renamings?
Renamings
    Renaming
    Renamings Renaming
Renaming
    with ( Identified as Identifier ) // TODO: comma separated list
    without ( Identifier ) // TODO: comma separated list

TraitPart :      // by analogy to PropertyAssignment
    PropertyName : AssignmentExpression
    get PropertyName() { FunctionBody }
    set PropertyName( PropertySetParameterList ) { FunctionBody }
    MethodDeclaration
    RequirementDeclaration
MethodDeclaration :
    method PropertyName ( FormalParameterList? ) { FunctionBody }
RequirementDeclaration :
    require PropertyName
```

Dynamic Semantics

Rather than define the semantics of this traits notation in terms of an existing library, we define internal functions that provide semantics equivalent to that provided by the traits library:

Internal Function	Corresponding Traits Library Method
TCreate	Trait.create
TCompose	Trait.compose
TResolve	Trait.resolve
TOverride	Trait.override

TODO: write a self contained description of the semantics of these internal functions that does not depend on the traits library.

A TraitLiteral is much like an ObjectLiteral. Notice that we could now define ObjectLiteral in terms of an expansion to a call to `Object.create` with a literal property-descriptor-map as second argument (ignoring infinite regress). Likewise, we define TraitLiteral in terms of an expansion to an expression that will generate the property-descriptor-map to be used as the second argument to `TCreate`.

We define the dynamic semantics of the above production as expansions to calls to these internal functions.

Syntax	Dynamic Semantics
`trait class T(x,y) ⇒{...}`	`const T(x, y) { return /*Trait Literal Expansion*/; }`
`class C(x,y) ⇒{...}`	`const C(x,y) { return TCreate(C.prototype, (trait class (x,y)⇒{...})(x,y)); }`
Trait Literal Expansion	
`{ /*mixins*/, /*parts*/ }`	`TOverride({ /*eparts*/ }, TCompose( /*emixins*/ ))`
`mixin T2(y,x)`	`T2(y,x)`
`mixin T2(y,x) with (p as q)`	`TResolve({ p: 'q' }, T2(y,x))`
`mixin T2(y,x) without (p)`	`TResolve({ p: undefined }, T2(y,x))`
`p: valExpr`	`p: {value: valExpr, enumerable: true}`
`get p() {body;}`	`p: {get: const() {body;}, enumerable: true}`
`set p(n) {body;}`	`p: {set: const(n) {body;}, enumerable: true}`
`method p(x,y) {body;}`	`p: {value: const(x,y) {body;}, method: true}`
`require p`	`p: {required: true}`

The "`/*emixins*/`" and "`/*eparts*/`" above are the expanded forms of "`/*mixins*/`" and "`/*parts*/`" according to the rest of the above table. If a `get` and a `set` appear for the same property, we combine their expansion into the obvious joint property descriptor rather than a conflict.

Like the FunctionDeclarations they expand to, ClassDeclarations and TraitClassDeclarations are hoisted to the beginning of the Block or Program they appear in, and so may be mutually recursive. Unlike FunctionDeclarations, by our "earlier error" rules below, these can throw a TypeError when evaluated, which is to say, on entry to their Block or Program. A ClassExpression or TraitClassExpression can also throw a TypeError when evaluated, but these are not hoisted so the error occurs according to where the expression appears.

Because of the algebraic properties of these traits operations, simpler TraitLiterals are equivalent to simpler expansions

Syntax	Simplified Dynamic Semantics
`{ /*mixins*/, /*parts*/ }`	`TOverride({ /*eparts*/ }, TCompose( /*emixins*/ ))`
`{ /*mixins*/ }`	`TCompose( /*emixins*/ )`
`{ /*mixin*/ }`	`/*emixin*/`
`{ /*parts*/ }`	`{ /*eparts*/ }`

Notice that overriding *only* occurs when a class or trait class uses both mixins and parts in one TraitLiteral.

Semi-Static Semantics

Earlier Errors

In addition to the expansion shown above, a TraitClassDeclaration and TraitClassExpression also records that the function it evaluates to (or binds to the trait class name) is in fact a trait class. Nothing other than a trait class may claim to be a trait class. However, we should not use [[Class]] to record this, as trait classes are still proper functions and must continue to appear to be proper functions. For example, the equivalent of `hiddenWeakMap.set(T, true)` would be an adequate record, as long it occurs before `T` is observable and the explanatory `hiddenWeakMap` is shared across frames (JavaScript global contexts).

When a class or trait class contains a TraitMixin, the Identifier in the TraitMixin (T2 above) is supposed to evaluate to a trait class. This identifier must be free in the class or trait class that this TraitMixin appears in, and must be declared by a TraitClassDeclaration or ConstDeclaration outside this class or trait class, so that the value it designates will be stable over the life of the individual class or trait class that mixes it in. If not, a static SyntaxError is reported.

When the class or trait class declaration or expression is evaluated, we also check that these captured values are indeed trait classes, i.e., a function that a TraitClassDeclaration or TraitClassExpression has recorded is a trait class, and throw a `TypeError` if not. To help keep track of the distinction, we name all our examples trait classes with an initial upper case and a `Trait` suffix. When this convention is used consistently, it should reliably prevent these *"not a trait class"* TypeErrors from occurring dynamically.

By virtue of these rules, when a class or trait class declaration or expression is evaluated, the VM can tell exactly what property names it requires and provides. Therefore, on evaluating a ClassDeclaration or ClassExpression, the VM also knows whether there are any conflicts or unresolved requirements. If so, a TypeError must be thrown then. Note that this is earlier than the equivalent reports from `Trait.create`, which do not happen until instances are made.

Because classes and trait classes are generative, these TypeErrors cannot in general be reported as static errors, but must wait until the class or trait class declaration or expression is evaluated. This is what we mean by "Semi-Static". For example, there's no way to tell statically whether the following code would throw these TypeErrors:

```
const makeEnhancedPointClass(const ExtraTrait) {
  trait class PointTrait(x, y) => {
    method getX() { return x; },
    method getY() { return y; }
  }
  class Point(x, y, color) => {
    mixin ExtraTrait(color),
    mixin PointTrait(x, y)
  }
  return Point;
}
```

It is not yet clear how useful the generality of the above pattern is, although I believe Newspeak demonstrates the utility of a similiar notion of parameterizable superclasses. (TODO: citation) In any case, the dynamic and generative nature of JavaScript argues for preserving this flexibility. As shown by the analysis here, this flexibility does not impede either our optimization or early diagnostic goals. When ClassDeclarations or TraitClassDeclarations appear at top level in a Program these errors are reported as early as possible anyway, when the Program is first evaluated and (because of hoisting) before any code in the Program runs.

Optimization Opportunities

By the same bookkeeping that allows these earlier errors to be reported, the VM can keep track of the method code associated with the provided method properties of each class or trait class. For each of these, an instance must *appear* to have the implied frozen bound methods as *own* properties. But so long as the properties are only ever invoked as methods of the instance, rather than read as observable property values, the frozen bound method object never need be created. Rather, for each class and trait class, the VM should also keep track

- statically, of the names of the variables used freely in its TraitLiteral, and
- dynamically, of the values bound to these names in the scope in which the TraitLiteral is evaluated.

Together, this pair is like the optimized representation of a [[Scope]] object used to implement storage-efficient closures.

The set of [[Scope]]s gathered together by a class is allocated as a hidden [[State]] field on the instance, indexed by the trait class of origin. As with lexical [[Scope]]s, the [[State]] has the same shape for all instances of the same class, and so can be allocated at the same time as the instance's non-method properties. When a method is invoked, the method code is evaluated in the context of the [[Scope]] for that method's trait within the instance's [[State]]. In addition to its explicit arguments, the method's code only needs be given access to the instance and be informed about the offset into this instance at which it should find its captured [[Scope]] variables. This offsetting logic should resemble C++'s

implementation of multiple inheritance.

Each instance also needs dynamic storage to memoize all the bound frozen methods which do get reified by explicit [[Get]] operations outside the context of a method invocation. Further [[Get]]s must check this memo first, or we would lose ===.

Again, because classes and trait classes are generative, these optimizations cannot in general be performed statically, but must wait until the class or trait class declaration or expression is evaluated. However, this still amortizes the optimization over all instances of these classes, which is the whole point. Again, for ClassDeclarations and TraitClassDeclarations appearing at top level in a Program, these optimizations occur as early as possible anyway, effectively statically at compile time.

Examples

Colored Point

The colored point example from the http://traitsjs.org front page, with minor revisions for Harmony without traits:

```
const makeColorTrait(col) {
  return Trait({
    color: function() { return col; }
  });
}
const makePoint(x, y) {
  return Trait.create(      // create an instance of a trait
    Object.prototype,        // that inherits from Object.prototype
      Trait.compose(           // and is the composition of
        makeColorTrait('red'), // a color trait
        Trait({                // and an anonymous point trait
          getX: const() { return x; },
          getY: const() { return y; },
          toString: const() { return ''+x+'@'+y; }
        })));
}
const p = makePoint(0,2);
p.color() // 'red'
```

The code above rewritten using this proposed notation:

```
trait class ColorTrait(col) => {
  method color() { return col; }
}
trait class PointTrait(x, y) => {
  method getX() { return x; },
  method getY() { return y; },
  method toString() { return ''+x+'@'+y; }
}
class Point(x, y) => {
  mixin ColorTrait('red'),
  mixin PointTrait(x, y)
}
const p = Point(0,2);
p.color(); // 'red'
```

A minor semantic difference is that our `Point` class makes instances that inherits from `Point.prototype` so that `p`

`instanceof Point` will return true. Whereas the original makes points that inherit directly from `Object.prototype`.

Unfortunately, this example also demonstrates a hazard created by the notational choices proposed above. To preserve the robustness of the original code, we had to pay the verbosity cost of introducing an intermediate `PointTrait` trait. Had we used the following more compact and legible code, the properties introduced by the `Point` class directly would silently override any conflicting properties from the traits being mixed in.

```
trait class ColorTrait(col) => {
  method color() { return col; }
}
class Point(x, y) => {
  mixin ColorTrait('red'),
  method getX() { return x; },
  method getY() { return y; },
  method toString() { return ''+x+'@'+y; }
}
const p = Point(0,2);
p.color(); // 'red'
```

Enumerable Trait

The Traits tutorial examples rewritten using out proposed notation.

`EnumerableTrait` itself is not improved by our new notation:

```
trait class EnumerableTrait() => {
  require forEach,
  method map(fun) {
    const seq = [];
    this.forEach(function(e,i) {
      seq.push(fun(e,i));
    });
    return seq;
  },
  method filter(pred) {
    const seq = [];
    this.forEach(function(e,i) {
      if (pred(e,i)) {
        seq.push(e);
      }
    });
    return seq;
  },
  method reduce(init, fun) {
    const result = init;
    this.forEach(function(e,i) {
      result = fun(result, e, i);
    });
    return result;
  }
};
```

`makeInterval` comes out both better and worse, since we again need to introduce an intermediate trait class to avoid the implied override:

```
trait class IntervalTrait(min, max) => {
  start: min,
  end: max,
  size: max - min - 1,
  method toString() { return ''+min+'..!'+max; },
  method contains(e) { return (min <= e) && (e < max); },
  method forEach(consumer) {
    for (let i = min; i < max; i++) {
      consumer(i,i-min);
    }
  }
}
class Interval(min, max) => {
  mixin EnumerableTrait(),
  mixin IntervalTrait(min, max)
}
const i = Interval(0,5);
i.start // 0
i.end // 5
i.reduce(0, const(a,b) { return a+b; }) // 0+1+2+3+4 = 10
```

These examples suggest that perhaps our syntax should implicitly compose where it currently implicitly overrides. This is doable, but leaves open the question of how to syntactically express an override.

Acks

Traits have a long history that we will not try to credit here. See http://traitsjs.org for the derivations of these ideas.

The Traits library itself is the basis for everything here, and is by Tom Van Cutsem with help from MarkM.

The syntax proposed above is primarily by MarkM, with influence by many public discussions on es-discuss, Allen Wirf-Brock's obj initialiser constructors proposal, and by Brendan's challenge, and some early feedback from Tom. The syntax here also borrows some ideas privately communicated to MarkM by Erik Aarvidson and Alex Russell.

The optimization ideas presented here are inspired by Ihab Awad and Mike Stay's old Cajita Optimization proposal.

Newspeak, for emphasizing the utility of parameterizable superclasses (TODO: cite needed).

See

http://traitsjs.org

Old Cajita Optimization proposal.

obj initialiser constructors