ECMAScript Proposal: First-Class Protocols

Michael Ficarra • 60th Meeting of TC39

Inspired by...

- Haskell type classes
- Rust traits
- Java 8+ interfaces
- Ruby mixins
- This pattern:
  `class A extends mixin(SuperClass, FeatureA, FeatureB) { /* … */ }`

PLEASE HOLD ALL SYNTAX DISCUSSION UNTIL AFTER STAGE 1 CONSENSUS

(I know you're not going to, but please try)

Simple Example: Foldable

```
protocol Foldable {
  foldr;

  toArray() {
    return this[Foldable.foldr](
      (m, a) => [a].concat(m), []);
  }

  get length() {
    return this[Foldable.foldr](
      m => m + 1, 0);
  }

  contains(eq, e) {
    return this[Foldable.foldr](
      (m, a) => m || eq(a, e),
      false);
  }
}
```

So what does this do?

Foldable is an object that looks mostly like this:

```
const Foldable = Object.assign(Object.create(null), {
  foldr: Symbol('Foldable.foldr'),
  toArray: Symbol('Foldable.toArray'),
  length: Symbol('Foldable.length'),
  contains: Symbol('Foldable.contains'),
});
```

Simple Example: Non-Empty List

```
class NEList implements Foldable {
  constructor(head, tail) {
    this.head = head;
    this.tail = tail;
  }

  [Foldable.foldr](f, memo) {
    if (this.tail != null) memo =
      this.tail[Foldable.foldr](
        f, memo);
    return f(memo, this.head);
  }
}
```

So what does this do?

- Defines a class in the usual way
- Checks for complete implementation of required protocol symbols (Foldable.foldr in the example)
- Copies methods from protocol into class prototype
- Copies static methods from protocol into class constructor

Calling Patterns: Explicit

```
let a = new NEList(1, null);
let b = new NEList(0, a);

b[Foldable.toArray]();
```

Calling Patterns: Aliased

```
let a = new NEList(1, null);
let b = new NEList(0, a);

const toArray = Foldable.toArray;

b[toArray]();
```

Calling Patterns: Functional

```
let a = new NEList(1, null);
let b = new NEList(0, a);

function toArray(x) {
  return x[Foldable.toArray]();
}

toArray(b);
```

Calling Patterns: Bind Operator

```
let a = new NEList(1, null);
let b = new NEList(0, a);

function toArray() {
  return this[Foldable.toArray]();
}

b::toArray();
```

Extending Existing Constructors: Protocol.implement

All of the upsides of extending built-in prototypes, none of the downsides!

```
protocol Functor { map; }

class Identity {
  constructor(val) { this.val = val; }
  unwrap() { return this.val; }
}

Promise.prototype[Functor.map] =
  function (f) {
    return this.then(function(x) {
      if (x instanceof Identity)
        x = x.unwrap();
      return new Identity(
        f.call(this, x));
    });
  };

Protocol.implement(Promise, Functor);
```

Protocol Inheritance

```
protocol A { a; }
protocol B extends A { b; }

class C implements B {
  [A.a]() {}
  [B.b]() {}
}

class D implements A {
  [A.a]() {}
}
```

Querying: the implements operator

```
protocol I { a; b() {} }
protocol K { a; b() {} }

class C {
  [I.a]() {}
}
C implements I; // false
C implements K; // false

class D implements I {
  [I.a]() {}
  [K.a]() {}
}
D implements I; // true
D implements K; // false
```

Combined Export Form

```
// module-a.js
protocol I { a; b(){} }
export { I };

export protocol K { a; b(){} }

// module-b.js
import { I, K } from './module-a.js'
class C implements I, K {
  [I.a]() {}
  [K.a]() {}
}
```

Protocol API

```
const Foldable = new Protocol({
  name: 'Foldable',
  extends: [ … ],
  symbols: {
    foldr: Symbol('Foldable.foldr'),
  },
  staticSymbols: { … },
  protoProperties:
    Object.getOwnPropertyDescriptors({
      toArray() { … },
      get length() { … },
      contains(eq, e) { … },
    }),
  staticProperties: …,
});
```

Example: backward compatible first-class iteration protocol

```
const Iterable = new Protocol({
  name: 'Iterable',
  symbols: {
    iterator: Symbol.iterator
  },
  protoProperties:
    Object.getOwnPropertyDescriptors({
      forEach(f) {
        let i = 0;
        for (let entry of this) {
          f.call(this, entry, i, this);
          ++i;
        }
      },
    }),
});
```

Example: backward compatible first-class iteration protocol

(cont)

```
Protocol.implement(Array, Iterable);
Array implements Iterable; // true

class NEList implements Iterable {
  constructor(head, tail) {
    this.head = head;
    this.tail = tail;
  }

  *[Iterable.iterator]() {
    yield this.head;
    if (this.tail != null) {
      yield* this.tail
        [Iterable.iterator]();
    }
  }
}

NEList implements Iterable; // true
```

Play with it!

https://github.com/disnet/sweet-interfaces

sweet.js implementation

comprehensive test suite

Open Questions

- Should protocols inherit from Object.prototype?
- Immutable prototype exotic objects?
- Frozen/Sealed?
- Should we have some Protocol.prototype?
- Actually copy symbols to prototype / constructor or use internal slots for resolution?
- How do `super` properties and `super` calls work?
- How does this have to interact with the global symbol registry?
- Should methods be created in realm of implementor or just once in realm of definition?

Stage 1?

syntax bike-shedding time

(or just save your comments for GitHub)