Tradeoffs in language design: The case of Javascript proxies

Tom Van Cutsem
(joint work with Mark S. Miller, with feedback from many others)
What do these have in common?
What do these have in common?

VIRTUALIZATION

Meta-object protocols are about virtualizing objects.
Virtualizing objects

querying an object
acting upon an object
“introspection”

```
obj["x"]
delete obj.x
"x" in obj
```

Reflection entails two types of API. Introspection API is the more common one.
Virtualizing objects

Implementing an object?

"Intercession"

```plaintext
obj["x"]
delete obj.x
"x" in obj
```

Intercession gives programmer the ability to *define* the behavior of an object in response to language-level operations like property access, etc.
Why implement your own objects?

- **Generic wrappers** around existing objects: access control wrappers (security), tracing, profiling, contracts, taint tracking, decorators, adaptors, ...

![Diagram: Proxy and Target](proxy_target.png)

- **Virtual objects**: remote objects, mock objects, persistent objects, promises / futures, lazily initialized objects, ...

![Diagram: Proxy and Question](proxy_question.png)

Proxies have many use cases. Can roughly categorize them based on whether the proxy wraps another target object in the same address space.
The Javascript object zoo

Native objects
(provided by ECMAScript engine)

Normal objects
(implementable in Javascript)

Host objects
(provided by the embedding environment, usually the browser)

The “DOM”

an Object

an Array

a NodeList

The “Virtual objects” category formed a major motivation for adding proxies to Javascript.
Why? Javascript scripts interact with different “types” of objects. Array and NodeList look and feel like normal JS objects, but differ from them in powerful ways (e.g. magical “length” data property)
With proxies, we can self-host an entire JS environment. E.g. a virtualized DOM. We can implement Array and NodeList in Javascript itself.
We started out with an API focused around the virtual objects use case, but then got into trouble when virtualizing invariants, then designed a new API focused on wrapping objects.
Example: revocable references

- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy

var {proxy, revoke} = makeRevocable();
// pass out proxy to third-party code
(proxy)
// ... later, can revoke the proxy
revoke();

A simple example of a Proxy abstraction.
Example: revocable references

- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy

```javascript
var {proxy, revoke} = makeRevocable(resource);
```

![Diagram showing plugin, proxy, and resource with revoke operation]
Example: revocable references

- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy

```javascript
var {proxy, revoke} = makeRevocable(resource);

plugin.give(proxy)
```
Example: revocable references

- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy

```javascript
var {proxy, revoke} = makeRevocable(resource);

plugin.give(proxy)
...
revoke();
```
function makeRevocable(target) {
    var enabled = true;
    var proxy = Proxy({
        proxy: proxy,
        revoke: function() { enabled = false; }
    })
    return {
        proxy: proxy,
        revoke: function() { enabled = false; }
    }
}
function makeRevocable(target) {
    var enabled = true;
    var proxy = Proxy({
        get: function(rcvr, name) {
            if (!enabled) throw Error("revoked")
            return target[name];
        },
        set: function(rcvr, name, val) {
            if (!enabled) throw Error("revoked")
            target[name] = val;
        },
        ...
    });
    return {
        proxy: proxy,
        revoke: function() { enabled = false; }
    }
}
function makeRevocable(target) {
    var enabled = true;
    var proxy = Proxy({
        get: function(rcvr, name) {
            if (!enabled) throw Error("revoked")
            return target[name];
        },
        set: function(rcvr, name, val) {
            if (!enabled) throw Error("revoked")
            target[name] = val;
        },
        ...
    });
    return {
        proxy: proxy,
        revoke: function() { enabled = false; }
    }
}
Stratified API

```javascript
var proxy = Proxy(handler);
```

Note: proxy.get does not reveal the handler trap. This is different from Spidermonkey’s __noSuchMethod__, Smalltalk’s doesNotUnderstand:, Ruby’s method_missing.
Stratified API

var proxy = Proxy(handler);

handler.get(proxy, 'foo')

Note: proxy.get does not reveal the handler trap.
This is different from Spidermonkey’s __noSuchMethod__, Smalltalk’s doesNotUnderstand:, Ruby’s method_missing.
var proxy = Proxy(handler);

handler.get(proxy, 'foo')
handler.set(proxy, 'foo', 42)

proxy.foo
proxy.foo = 42

Note: proxy.get does not reveal the handler trap. This is different from Spidermonkey’s __noSuchMethod__, Smalltalk’s doesNotUnderstand:, Ruby’s method_missing.
Stratified API

```javascript
var proxy = Proxy(handler);

handler.get(proxy, 'foo')
handler.set(proxy, 'foo', 42)
handler.get(proxy, 'get')

proxy.foo
proxy.foo = 42
proxy.get
```

Note: proxy.get does not reveal the handler trap.
This is different from Spidermonkey’s __noSuchMethod__, Smalltalk’s doesNotUnderstand:, Ruby’s method_missing.
Not just property access...

```javascript
var proxy = Proxy(handler);
```
Not just property access...

```javascript
var proxy = Proxy(handler);

handler.has('foo')
```

![Diagram](image)
Not just property access...

```
var proxy = Proxy(handler);

handler.has('foo')

handler.delete('foo')

...

'foo' in proxy

delete proxy.foo

...
```
... but not quite everything either

```javascript
var proxy = Proxy(handler);
```

Proxies have their own object identity. Can’t spoof the identity of another object.
Frozen objects (new since ECMAScript 5)

```javascript
var point = { x: 0, y: 0 };
Object.freeze(point);

point.z = 0;       // error: can’t add new properties
delete point.x;   // error: can’t delete properties
point.x = 7;      // error: can’t assign properties

Object.isFrozen(point) // true
```

guarantee (invariant):
properties of a frozen object are immutable

freezing is permanent - there is no defrost

Before ES5: could not build robust object abstractions that could be reliably shared between multiple third-party clients. The client could just mutate the object. With frozen objects, JS objects can acquire strong invariants.
How to combine proxies with frozen objects?

- Can a proxy emulate the “frozen” invariant of the object it wraps?

```javascript
var point = { x: 0, y: 0 };
Object.freeze(point);

var {proxy, revoke} = makeRevocable(point);

Object.isFrozen(point) // true
Object.isFrozen(proxy) // ?
```

Not clear how the proxy can acquire the “frozen state” of the object it wraps.
How to combine proxies with frozen objects?

• Can a proxy emulate the “frozen” invariant of the object it wraps?

```javascript
function wrap(target) {
    return Proxy({
        get: function(rcvr, name) { return Math.random(); };
    });
}

var point = { x: 0, y: 0 };
Object.freeze(point);

var proxy = wrap(point);

Object.isFrozen(point) // true
Object.isFrozen(proxy) // can’t be true!
```

We don’t know if proxy is frozen. That depends on the behavior of the proxy handler.
The “Solution”

- Proxies can’t emulate frozen objects
- `Object.isFrozen(proxy)` always returns `false`
- Safe, but overly restrictive
Language Design Tradeoff

Powerful proxies that can virtualize frozen objects

Strong language invariants that can’t be spoofed
Second iteration: “direct” proxies

- Proxy now has direct pointer to target: `Proxy(target, handler)`
- `Object.isFrozen(proxy) <=> Object.isFrozen(target)`

Second version of the Proxy API is very similar to the first, except the proxy now has a direct reference to a “target” object that it wraps.
Revocable references (old API)

```javascript
function makeRevocable(target) {
    var enabled = true;
    var proxy = Proxy({
        get: function(rcvr, name) {
            if (!enabled) throw Error("revoked")
            return target[name];
        },
        set: function(rcvr, name, val) {
            if (!enabled) throw Error("revoked")
            target[name] = val;
        },
        ...
    });
    return {
        proxy: proxy,
        revoke: function() { enabled = false; }
    }
}
```
Revocable references (new API)

function makeRevocable(target) {
    var enabled = true;
    var proxy = Proxy(target, {
        get: function(tgt, name) {
            if (!enabled) throw Error("revoked")
            return target[name];
        },
        set: function(tgt, name, val) {
            if (!enabled) throw Error("revoked")
            target[name] = val;
        },
        ...}
    );
    return {
        proxy: proxy,
        revoke: function() { enabled = false; }
    }
}
function makeRevocable(target) {
  var enabled = true;
  var proxy = Proxy(target, {
    get: function(tgt, name) {
      if (!enabled) throw Error("revoked")
      return target[name];
    },
    set: function(tgt, name, val) {
      if (!enabled) throw Error("revoked")
      target[name] = val;
    },
    ...
  });
  return {
    proxy: proxy,
    revoke: function() { enabled = false; }
  }
}
A proxy for a frozen object is itself frozen. But how can we be sure a frozen proxy actually *behaves* like a frozen object?
Direct proxies for frozen objects perform runtime assertions, checking whether the result returned by the trap corresponds to the frozen object’s state. In this example: property access on a frozen object should always return the same result.
Summary: tradeoffs in language design

- No free lunch:
  - Direct proxies are more complicated (invariant checks)
  - The two Proxy APIs support dual use cases. But: having both virtual and direct proxies in the language further increases complexity.