Ecma/TC39/2012/007

A

On the design of the ECMAScript Reflection API

TOM VAN CUTSEM, Vrije Universiteit Brussel
MARK S. MILLER, Google Research

We describe in detail the new reflection API of the upcoming Javascript standard. The most prominent feature of this new API is its support for creating proxies: virtual objects that behave as regular objects, but whose entire "meta-object protocol" is implemented in Javascript itself. Next to a detailed description of the API, we describe a more general set of design principles that helped steer the API's design, and which should be applicable to similar APIs for other languages. We also describe access control abstractions implemented in the new API, and provide an operational semantics of an extension of the untyped λ-calculus featuring proxies.

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

General Terms: Design, Languages

Additional Key Words and Phrases: Proxies, Javascript, reflection, intercession, membrane

1. INTRODUCTION

We introduce the upcoming reflection API for Javascript, in which dynamic proxies play a major role. Proxies have a wide array of use cases [Eugster 2006]. Two general cases can be distinguished depending on whether the proxy is proxying another object within the same address space:

Generic wrappers. Proxies that wrap other objects in the same address space. Example uses include access control wrappers (e.g. revokable references), higher-order contracts [Findler and Felleisen 2002], profiling, taint tracking, etc.
Virtual objects. Proxies that emulate other objects, without the emulated objects having to be present in the same address space. Examples include remote object proxies (emulate objects in other address spaces), persistent objects (emulate objects stored in databases), transparent futures (emulate objects not yet computed), lazily instantiated objects, test mock-ups, etc.

In previous work [Van Cutsem and Miller 2010], we designed a Javascript Proxy API that was centered around virtual objects, driven primarily by the use case of virtualizing the Document Object Model (DOM), the interface between Javascript scripts and the browser environment. This work directly builds upon that work, but updates and extends it as follows:

(1) Our previous API disallowed proxies to emulate objects with strong invariants (e.g. objects purporting to contain immutable properties). In this work, we introduce an updated API that overcomes these limitations. Interestingly, it does so by focussing the API on the generic wrappers use case, rather than the virtual objects use case. However, virtual objects can still be implemented in our updated API, just like generic wrappers could be implemented using our previous API.
(2) We describe a technique called *invariant enforcement* that enables Proxy APIs based on *wrapping* existing objects to uphold any language-specific invariants of the wrapped object. The principles behind invariant enforcement should be generally applicable to languages other than Javascript.
(3) We provide an operational semantics of an extension of the untyped lambda calculus including proxies, inspired by a subset of Javascript.
(4) This paper provides a more complete and more up-to-date description of the Javascript Proxy API and its current status.

The contributions of this paper are first, a detailed description of the new standard reflection API for the upcoming edition of Javascript (Section 4), second, the enumeration of the important design principles that helped guide our API, and which are applicable to message-based object-oriented reflection APIs in general (Section 5), third, a description of concrete access control wrappers implemented using our API (Section 7) and fourth, an operational semantics of the key aspects of our Proxy API (Section 8).

2. REFLECTION TERMINOLOGY

A metaobject protocol (MOP) [Kiczales et al. 1991] is an object-oriented framework that describes the behavior of an object-oriented system. It is a term most commonly associated with reflective object-oriented programming languages. It is customary for a MOP to represent language operations defined on objects as method invocations on their meta-objects. Throughout the rest of this paper, we use the general term *operation* to denote mechanisms such as message sending, field access and assignment, defining a method, performing an `instanceof` operation, and so on.

According to Kiczales *et. al* [Kiczales et al. 1991] a MOP supports *introspection* if it enables reflective read-only access to the structure of an object. It supports *self-modification* if it is possible to modify this structure. Finally, it supports *intercession* if it enables programmers to redefine the semantics of operations on specific objects. Introspection is typically supported by all reflection APIs. Self-modification is more exceptional, and reflection APIs with extensive support for intercession are rare in well-known languages, the CLOS MOP being a notable exception (see Section 10.6).

We will use the term *intercession API* to refer to any API that enables the creation of new base-level objects with custom meta-level behavior.

3. JAVASCRIPT

Javascript is a scripting language whose language runtime is often embedded within a larger execution environment. By far the most common execution environment for Javascript is the web browser. While the full Javascript language as we know it today has a lot of accidental complexity as a side-effect of a complex evolutionary process, at its core it is a fairly simple dynamic language with first-class lexical closures and a concise object literal notation that makes it easy to create one-off anonymous objects. This simple core is what Crockford refers to as “the good parts” [Crockford 2008].

The standardized version of the Javascript language is named ECMAScript. The Proxy API described in this paper was designed based on the latest standard of the language, ECMAScript 5 [ECMA International 2009]. Because ECMAScript 5 adds a number of important features to the language that have heavily influenced our Proxy API, we briefly summarize the new features relevant to our discussion in the following section.

3.1. ECMAScript 5

ECMAScript 5 defines a new object-manipulation API that provides more fine-grained control over the nature of object properties [ECMA International 2009]. In Javascript, objects are records of *properties* mapping names (strings) to values. A simple two-dimensional diagonal point can be defined as:

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

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[1]). A non-configurable, non-writable data property is in essence a constant binding. 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 (Javascript uses object-based inheritance, further discussed in Section 4.3). Its second argument is an object mapping property names to property descriptors. We could have also defined the `point` object explicitly as:

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

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*. By default, Javascript objects are extensible collections of properties. However, 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 data properties are all non-writable. The call `Object.freeze(obj)` freezes the object `obj`, effectively making the structure of `obj` (but not `obj`'s property values) immutable. Section 5.2 details how tamper-proof objects have influenced the design of our intercession API.

3.2. Reflection in Javascript

Javascript has built-in support for introspection and self-modification. These features are provided as part of the language, rather than through a distinct metaobject protocol. This is largely because Javascript objects are represented as flexible records map-

[1] With the exception that a non-configurable, writable data property can still be made non-writable.

ping strings to values. Property names can be computed at runtime and their value can be retrieved using array indexing notation. The following code snippet demonstrates introspection and self-modification:

```
var o = { x: 5, m: function(a) {...} };
// introspection:
o["x"]                  // computed property access
"x" in o                // property lookup
for (prop in o) {...}   // property enumeration
o["m"].apply(o,[42])    // reflective method call
// self-modification:
o["x"] = 6              // computed property assignment
o.z = 7                 // add a property
delete o.z              // remove a property
```

The new property descriptor API discussed in the previous section provides for more fine-grained introspection and self-modification of Javascript objects, as it additionally reveals the property attributes.

3.3. Intercession in Javascript

The current Javascript standard has no support for intercession. It is not possible to intercept an object's property access, how it responds to the `in`-operator, `for-in` loops and so on. Mozilla's Spidermonkey engine has long included a non-standard way of intercepting method calls based on Smalltalk's `doesNotUnderstand:` method (see Section 10.1). In Spidermonkey, the equivalent method is named `__noSuchMethod__`. For example, a proxy that can generically forward all received messages to a target object `o` is created as follows:

```
function makeProxy(target) {
  return {
    __noSuchMethod__: function(name, args) {
      return target[name].apply(target, args);
    }
  };
}
```

Throughout the rest of this paper, we will refer to methods that intercept language-level operations as *traps*, a term borrowed from the Operating Systems community. Methods such as `__noSuchMethod__` and `doesNotUnderstand:` are traps because they intercept method calls.

The problem with `doesNotUnderstand:` and derivatives is that the trap is not *stratified*: it is defined in the same name space as the rest of the application code. The only way in which a trap is distinguished from a regular method is by its name. This violation of stratification can lead to confusion:

— Say an object intentionally defines the trap to intercept invocations. Since the trap is part of the object's interface, its clients can accidentally invoke the trap as if it were an application-level method. This confuses meta-level code, since the call "originated" from the base-level.
— Say an object accidentally defines an application-level method whose name matches that of the trap. The VM will then incidentally invoke the application method as if it were a trap. This confuses the base-level code, since the call "originated" from the meta-level.

Without stratification, the intercession API pollutes the application's namespace. We conjecture that this lack of stratification has not posed a significant problem in practice because systems such as Smalltalk and Spidermonkey define only *one* such special

method. But the approach does not scale. If we were to introduce additional such traps to intercept not only method invocation, but also property access, assignment, lookup, enumeration, etc. the number of “reserved method names” on objects would quickly grow out of control.

From this and the previous section, it is clear that Javascript is an extremely dynamic language in which a number of core language features (computed property access, `for-in` loops) would actually be classified as reflective features in other languages. The tight, mostly unstratified, interweaving of core and reflective features is one motivation for introducing a more principled, stratified reflection API in Javascript.

Since Javascript has multiple operations defined on objects (e.g. property assignment and enumeration via `for-in` loops) and since `__noSuchMethod__` is only able to intercept one of these (method invocation), there is currently no mechanism in Javascript to create fully transparent wrappers. For instance, enumerating the properties of a proxy generated by the `makeProxy` function above would enumerate the properties of the empty proxy object, rather than those of its target object. As we describe next, a more faithful wrapping or emulation of objects is sometimes necessary.

3.4. Javascript Host Objects

Another motivator for the introduction of scriptable proxies in Javascript is the existence of so-called *host objects*. Host objects are objects implemented by the host platform (for instance, the browser). To ordinary Javascript objects, these host objects (mostly) behave like any other Javascript object. Most core browser APIs, such as the Document Object Model (DOM), are composed almost entirely out of such host objects. Typically, interaction with a host object leads to changes in the host platform (e.g. a reflow or repaint of the HTML document).

Since host objects are entirely provided by the host platform, and mostly implemented in a language other than Javascript, host objects have the freedom to behave arbitrarily differently from regular objects. For instance, over time, browser APIs like the DOM have exposed a number of host objects to Javascript code with behavior that cannot normally be expressed by Javascript objects. The problem here is that Javascript libraries cannot faithfully emulate or wrap such host objects.

Reasons for emulating these host objects range from implementing libraries that want to augment, or fix issues with, existing host objects, all the way to libraries that want to fully virtualize host environment APIs such as the DOM. The Proxy API discussed in the next section enables such virtualization.Indeed, the `dom.js` library, under active development, has the goal of fully virtualizing the DOM using our Proxy API [Gal and Flanagan 2011].

4. JAVASCRIPT PROXIES

As a general example of the kind of use cases that proxies enable, consider revokable references. Say an object `Alice` wants to hand out to `Bob` a reference to `Carol`. `Carol` could represent a precious resource, and for that reason `Alice` may want to limit the lifetime of the reference she hands out to `Bob`, which she may not fully trust. In other words, `Alice` wants to have the ability to revoke `Bob`’s access to `Carol`. Once revoked, `Bob`’s reference to `Carol` becomes useless.

One can implement this pattern of access control by wrapping `Carol` in a forwarding proxy that can be made to stop forwarding. This is also known as the *caretaker* pattern [Redell 1974]. Without proxies, the programmer is forced to write a distinct caretaker for each kind of object that should be wrapped. Proxies enable the programmer to abstract from the details of the wrapped object’s protocol and instead write a *generic* caretaker. Using such a generic caretaker abstraction, `Alice` can write:

```
var carol = {...};
// caretaker is a tuple consisting of a proxy reference, and a revoke function
var caretaker = makeCaretaker(carol);
// caretaker.ref is a proxy for carol, which alice can give to bob:
bob.use(caretaker.ref);
// later, alice can revoke bob's access...
caretaker.revoke();
// caretaker.ref is now useless
```

A key point is that as long as the caretaker is not revoked, `Bob` can use the proxy for `Carol` as if it were the real `Carol`. There is no need for `Bob` to change the way he interacts with `Carol`. Indeed, in the Proxy API we are about to present, if `Bob` has no other direct reference to `Carol`, and as long as the reference is not revoked by `Alice`, `Bob` is not even able to tell that `caretaker.ref` is only a proxy for `Carol`.

In Section 7.1 we show an implementation of the `makeCaretaker` function using the Proxy API discussed below.

4.1. The Proxy API

We now describe our Proxy API for Javascript. This API is slated for inclusion in the next ECMAScript standard[2].

Our Proxy API supports intercession by means of distinct *proxy* objects. The behavior of a proxy object is controlled by a separate handler object. The methods of the handler object are traps that are called whenever a corresponding operation is applied to the proxy object. Handlers are effectively “meta-objects” and their interface effectively defines a “metaobject protocol”. A proxy object is created as follows:

```
var proxy = Proxy(target, handler);
```

Here, `target` is an existing Javascript object that is going to be wrapped by the newborn proxy. `handler` is an object that may implement a particular meta-level API. Figure 1 depicts the relationship between these objects.

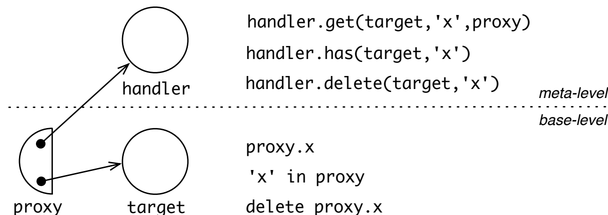

Fig. 1. Relationship between `proxy`, `target` and `handler`.

Table I lists those base-level operations applicable to objects that can be trapped by handlers. The distinction between fundamental and derived traps is explained in Section 6.1. Some operations come in different flavors, depending on whether they are expected to only look at the target’s “own” properties or whether they should also take into account properties inherited from the target’s prototype.

An intercepted operation mostly simply triggers the corresponding trap and returns its result. The `enumerate` trap must return an array of strings representing the enumerable property names of the proxy. The corresponding `for-in` loop is then driven by

[2]A draft specification is available at `http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies`.

Table I. Operations reified on `p = Proxy(t,h)`

Operation	Triggered by	Reified as
Fundamental traps		
Descriptor lookup	`Object.getOwnPropertyDescriptor(p,name)`	`h.getOwnPropertyDescriptor(t,name)`
Descriptor definition	`Object.defineProperty(p,name,pd)`	`h.defineProperty(t,name,pd)`
Own property listing	`Object.getOwnPropertyNames(p)`	`h.getOwnPropertyNames(t)`
Property deletion	`delete p[name]`	`h.deleteProperty(t,name)`
Preventing extensions	`Object.preventExtensions(p)`	`h.preventExtensions(t)`
Function application	`p(...args)`	`h.apply(t,undefined,args)`
Derived traps		
Property query	`name in p`	`h.has(t,name)`
Own property query	`({}).hasOwnProperty.call(p,name)`	`h.hasOwn(t,name)`
Property lookup	`p[name]`	`h.get(t,name,p)`
Property assignment	`p[name] = val`	`h.set(t,name,val,p)`
Property enumeration	`for (var name in p) {}`	`h.enumerate(t)`
Own property enum.	`Object.keys(p)`	`h.keys(t)`
Sealing	`Object.seal(p)`	`h.seal(t)`
Freezing	`Object.freeze(p)`	`h.freeze(t)`
Object construction	`new p(...args)`	`h.construct(t,args)`

iterating over this array. Because Javascript methods are just functions, a method invocation `proxy.m(a,b)` is reified as a property access `handler.get(target,"m",proxy)` that is expected to return a function. That function is immediately applied to the arguments `[a,b]` with its `this`-pseudovariable bound to `proxy`.

All traps in the above API are optional. If a handler does not define a trap, the proxy will forward the intercepted operation to its `target` unmodified. For instance, if `handler` does not define a `get` trap, then `proxy["x"]` is equivalent to `target["x"]`.

The distinction between proxy objects and regular objects ensures that non-proxy objects (which we expect make up the vast majority of objects in a typical heap) do not pay the runtime costs associated with intercession (cf. Section 9). Finally, the references that a proxy holds to its target and handler are immutable and inaccessible to clients of the proxy.

As an illustration of our API, consider a proxy wrapper that simply wants to log all property assignments performed on its wrapped target object, but otherwise does not want to change the behavior of the wrapped object:

```
function makeChangeLogger(target, log) {
  return Proxy(target, {
    set: function(target, name, value, receiver) {
      var success = Reflect.set(target, name, value, receiver);
      if (success) {
        log('property '+name+' on '+target+' set to '+value);
      }
      return success;
    }
  });
}
```

The `Reflect.set` method forwards the intercepted property assignment operation to the `target` object, returning whether or not the property was updated successfully. The `Reflect` object is discussed in more detail in Section 4.4.

4.2. Functions

In Javascript, functions are objects. However, they differ from non-function objects in a number of ways. In particular, functions support two operations not applicable to objects: function application and object construction. Construction is performed using the `new` operator: if `F` is a function, `new F()` creates a new object `o` whose prototype

object is `F.prototype`. `F` is then applied, with its `this`-pseudovariable bound to `o`, so that it can initialize the new object. Think of functions used in this manner as playing the role of constructors in class-based languages.

The `apply` and `construct` traps are only enabled when the wrapped `target` is itself a function. If this is not the case, calling `proxy(...args)` or `new proxy(...args)` results in an error, just like when trying to apply or construct a non-function value[3]. The following code snippet illustrates the interaction between functions and proxies. It defines a proxy `p` that wraps a function `f` and acts as a simple contract, enforcing that a) the function is only called with numbers and b) it is never used as a constructor.

```
var f = function(a,b) { return a+b; };
var h = {
  apply: function(target,receiver,args) {
    args.forEach(function(arg) { assert(Number(arg) === arg); });
    return Reflect.apply(target,receiver,args);
  },
  construct: function(target,args) {
    throw new Error("not a constructor");
  }
};
var p = Proxy(f,h);
f(1,2);      // returns 3
p(1,2);      // calls h.apply(f, undefined, [1,2]), returns 3
p('a','b');  // calls h.apply(f, undefined, ['a','b']), throws exception
new p(1,2);  // calls h.construct(f, [1,2]), throws exception
p.x          // get trap missing, defaults to f.x, returns undefined
var o = { m: p };
o.m(1,2);    // calls h.apply(f,o, [1,2]), returns 3
```

The `apply` trap receives as its second argument the value to use for the `this`-pseudovariable. When a function is applied without a receiver, as in the expression `p(1,2)`, this parameter is set to `undefined`. When a function is stored as a property of an object and invoked as a method, as in the expression `o.m(1,2)`, this parameter is set to the receiver object `o`.

4.3. Interaction with Prototype Inheritance

Javascript features object-based inheritance: every object has a prototype link that designates the object in which lookup should proceed, if a particular property is not found in the object itself. Any object may serve as a prototype for other objects. Proxies may thus also become the prototype of other objects.

Four of the operations outlined in Table I may climb the prototype chain: property query (`has`), property lookup (`get`), property assignment (`set`) and property enumeration (`enumerate`). When any of these operations hits a proxy while climbing a prototype chain, the ascent is stopped and control is transferred to the corresponding proxy trap.

The third parameter of the `get` and `set` traps identifies the initial receiver of the property access or assignment[4]. If the proxy itself was the initial receiver of the property access or assignment, this parameter will be the proxy itself. Otherwise, it refers to a "child" that inherits (directly or indirectly) from the proxy:

```
var proxy = Proxy(target, handler);
// child is an empty object inheriting from proxy
```

[3]The `...arg` notation is new syntax in the next version of ECMAScript, used to turn the elements of an array into actual arguments (when used at a call site), or to bind rest parameters (when used in formal parameter lists).

[4]These traps need access to that initial receiver to correctly bind the `this`-pseudovariable in inherited accessor properties.

```
var child = Object.create(proxy);
proxy[name];      // triggers handler.get(target,name,proxy)
child[name];      // triggers handler.get(target,name,child)
proxy[name] = 1;  // triggers handler.set(target,name,1,proxy)
child[name] = 1;  // triggers handler.set(target,name,1,child)
```

The ability to inherit from a proxy is often useful, e.g. to be able to intercept only missing properties: if `child` would define a property `name`, the proxy's `get` or `set` trap will not be triggered for that property. Only when a property does not exist in `child` is the proxy triggered.

4.4. Reflect: the dual of the Proxy API

When writing a Proxy handler, one is often interested in *augmenting* the default behavior of the target object in response to an intercepted operation. As such, a generic mechanism is needed by which the implementor of a handler trap can "forward" the intercepted operation to its target. Typical use cases coincide with what would be expressed as "before", "after" and "around" advice in aspect-oriented programming [Kiczales and Hilsdale 2001], or in CLOS using method combinations [Paepcke 1993].

The Proxy API provides a distinct `Reflect` object[5] which defines, for each trap in the Proxy API, a method with the same name and arguments. When invoked, the method applies the intercepted operation to its first parameter.

The `Reflect` object is the dual of a `Proxy` handler object: a proxy handler can uniformly *intercept* operations on an object, while the `Reflect` object can uniformly *perform* these operations on an object. The following code snippet illustrates this duality for the property query operation:

```
var proxy = Proxy(target, handler);
name in proxy              // equivalent to: handler.has(target, name)
Reflect.has(target, name)  // equivalent to: name in target
Reflect.has(proxy, name)   // equivalent to: name in proxy
                           //   and thus to: handler.has(target, name)
```

The `makeChangeLogger` example in Section 4.1 made use of `Reflect.set` to forward property assignment to the wrapped target object. We will make further use of the `Reflect` object in Sections 6.2 and 7.

5. DESIGN PRINCIPLES

In this section, we describe the main design principles that helped shape the Proxy API discussed in the previous section. These principles have merit and applicability beyond the particulars of our ECMAScript API, and should be applicable to any object-oriented intercession API built around proxies.

5.1. Stratification

Bracha and Ungar [Bracha and Ungar 2004] introduce the principle of stratification for mirror-based architectures. The principle states that meta-level facilities should be separated from base-level functionality. Bracha and Ungar focus mostly on stratification in the context of introspection and self-modification. In this paper we focus on the application of this principle to intercession. Mirrors are further discussed in Section 10.4.

The distinction between a proxy and its handler object enforces stratification of the traps. Traps are not defined as part of the (application-level) interface of the

[5] In the ECMAScript standard, this object will instead be defined as a module, but this detail is irrelevant here and one can think of the `Reflect` object as a first-class representation of the module.

proxy object, but as part of the interface of the handler. For instance, the property access `proxy.has` will not trigger the proxy's corresponding `has` trap. Instead, it will correctly trigger `handler.get(target,"has",proxy)`. Likewise, `proxy.get` triggers `handler.get(target,"get",proxy)`. Traps can only be invoked as methods on a proxy's handler, not on the proxy itself. This enforces stratification (the meta-level traps should not interfere with base-level method names). Thus, proxies continue to work correctly if an application (by accident or by design) uses the names `get`, `set`, `has`, etc.

Separating handler and proxy into separate objects has other benefits. The handler has its own prototype chain which is completely independent from that of its proxy. A single handler may also handle multiple proxies. Indeed, all handler traps are parameterized with the `target` to operate on, so that a shared handler can distinguish the different targets of its proxies. The handler can even be a proxy itself (we illustrate this in Section 6.2).

The principle of stratification when applied to proxies can be summarized as follows:

Stratification: by defining a proxy's traps on a separate handler object, the proxy's application-level interface remains cleanly separated from its meta-level interface.

5.2. Invariant Enforcement

Recall from section 3.2 that ECMAScript 5 enables the creation of tamper-proof objects. A tamper-proof object provides useful invariants that programmers can rely upon. When designing a Proxy API, care should be taken that proxies do not inadvertently break these invariants. ECMAScript 5 provides two important invariants:

Non-configurability. The attributes of a non-configurable property of an object cannot change over time. In particular, a non-writable, non-configurable data property is effectively a "constant" property. Also, a non-configurable property cannot be deleted.

Non-extensibility. Once an object `obj` is made non-extensible by a call to `Object.preventExtensions(obj)`, `Object.seal(obj)` or `Object.freeze(obj)`, new properties can no longer be added to it.

For example, the call `Object.freeze(obj)` makes `obj` non-extensible, makes all of its properties non-configurable and all data properties non-writable. The programmer can now make assumptions about the behavior of `obj` based on the above invariants (for instance, caching the value of `obj`'s data properties without requiring cache invalidation). If `obj` is a proxy, it should not be able to violate these assumptions. Invariants are important for humans to reason about code, critical for security, and useful for compilers and virtual machines.

In our earlier Proxy API design, we found that the easiest way to accomplish invariant enforcement was to simply never allow a proxy to emulate an object with the above invariants [Van Cutsem and Miller 2010]. However, this severely restricted the applicability of proxies. For instance, it precluded proxies from emulating host objects with non-configurable properties.

The reason why our previous Proxy API could not enforce invariants was that its API was designed to primarily implement virtual objects. These proxies did not directly wrap an existing target object, so it was not clear what invariants, if any, would need to be enforced on the virtual object. In our revised design, a proxy always wraps a known target object. Thus, the proxy can inspect its target to check its invariants (i.e. which properties are non-configurable, and whether or not the object is non-extensible). As such, whenever the handler intercepts an operation and produces a result, the proxy can *verify* whether the result is acceptable for the intercepted operation, given the target's invariants. If it is, the proxy reveals the result to clients. If not, it throws an exception, thereby notifying clients that the handler was about to violate an invariant.

As a concrete example of such invariant enforcement, consider the `deleteProperty` trap. Assume a property `foo` is deleted from a proxy object `p = Proxy(t,h)`, by evaluating `delete p.foo`. This triggers the `h.deleteProperty(t,"foo")` trap, which must return a boolean indicating whether the property was successfully deleted. After the proxy invokes the trap, it checks whether the handler returned `true`, which indicates a successful deletion. If so, it checks whether the `foo` property is declared as a non-configurable property of `t`. If it is, the proxy throws an exception. If the property is configurable or does not exist, or if the handler returned `false`, the outcome of the operation is reported to the client.

Our invariant enforcement mechanism for proxies hinges on the fact that the target object, which is queried for its invariants by the proxy, cannot lie about what invariants it upholds. We sketch an informal proof by induction on the target of a proxy.

First, we note that the chain of proxy-target links must be a finite, non-cyclic list: the target of a proxy must already exist before the proxy is created. It is not possible to initialize the target of a newborn proxy to that proxy itself.

In the base case, a proxy's target is a regular non-proxy object. Non-proxy objects by definition uphold language invariants, so the proxy can faithfully query the target for its invariants. Hence, the handler will not be able to violate reported invariants.

For the inductive step, consider a proxy a whose target is itself a proxy b. By the induction hypothesis, b cannot violate the invariants of its own target, so that a can faithfully query b for its invariants. Hence, a's handler will not be able to violate b's reported invariants.

The essence of the invariant enforcement mechanism can be summed up as follows:

Invariant Enforcement: to ensure that language invariants of a wrapped object cannot be violated by a proxy, have the proxy verify the results of its handler against the invariants of its target. This enables a reflection API to support intercession on objects with invariants, without weakening those invariants.

5.3. Selective Interception

As the previous Section illustrates, a proxy-based intercession API introduces a trade-off between what operations can be intercepted by proxy handlers on the one hand, versus what operations have a reliable outcome from the language runtime and the programmer's point of view. Invariant enforcement upholds certain language invariants, but for some operations even invariant enforcement would not be sufficient. Here, we consider reasons to prevent a proxy from intercepting certain operations entirely. The first is to preserve the stability of some operations, the second is to prevent unexpected interleaved execution of code.

5.3.1. Stable Operations. Some operations may be so critical that proxies should never be able to influence their outcome. One such operation is identity comparison. In Javascript, the expression `a === b` determines whether `a` and `b` refer to identical objects. The === operator comes with a number of implicit guarantees: it is commutative, transitive, symmetric and stable (it always reports the same answer given the same arguments). Furthermore, testing `a` and `b` for equality should not grant `a` access to `b` or vice versa. For all these reasons, our Proxy API does not allow proxy handlers to trap ===. Proxies cannot influence the outcome of this operation.

Stability is important for other operations as well. To uphold stability, the operation should not trap to a proxy handler, since a handler would be able to return different results over time. We describe three ways in which an operation can remain stable when applied to a proxy.

First, an operation may simply operate on proxies the same way it operates on non-proxies. The === operator remains stable on proxies in this way, since it uses the proxy's own distinct object identity.

Second, the stable outcome of an operation can be passed as a parameter to the `Proxy` constructor, when the proxy is created. The operation then simply returns this stable value directly as the result of the operation, without trapping the handler. Our previous design made use of this approach [Van Cutsem and Miller 2010].

Third, a stable operation can bypass a proxy and instead be performed on the proxy's target. If the operation is stable on regular, non-proxy objects, it will also be stable on proxies (the proof is analogous to our inductive proof of invariant enforcement). Our API has several examples of this kind:

— The built-in ECMAScript function `Object.getPrototypeOf(obj)` returns the prototype of the object `obj`. In standard ECMAScript, since the prototype link of an object is immutable, this is a stable operation. When `obj` is a proxy, its target's prototype link is returned instead. The proxy handler is not consulted.
— The ECMAScript operators `typeof obj` and `obj instanceof aFunction` are classification operators that cannot be intercepted by proxies. A proxy is classified the same way its target would be classified.
— The built-in ECMAScript function `Object.isExtensible(obj)` returns whether or not the object `obj` is extensible. An invariant is that once `Object.isExtensible(obj)` returns `false`, it should forever after return `false` (i.e. non-extensibility is a stable state). When `obj` is a proxy, the extensibility of the proxy's target is checked instead. Thus, a proxy cannot claim to be non-extensible at one point, and claim to be extensible at a later point in time.

5.3.2. Interleaved Execution. When designing a Proxy API, one has to take into account the fact that every intercepted language operation may trigger the execution of arbitrary code (in the handler trap). Since a Proxy API enables programmers to intercept what look like primitive operations, code can now be executed in places unexpected by the programmer or even the language runtime itself.

For example, in the absence of proxies (and host objects), the expression `name in obj` does not trigger any Javascript code. Since `in` is a primitive operation, the evaluation of this expression occurs entirely within the language runtime, without it ever having to execute non-native code. Proxies may change these assumptions, requiring programmers to think carefully about what invariants may be implicitly violated by proxies.

In the particular case of our Javascript Proxy API, the possibility of new interleaved code paths was not considered as critical for two reasons. First, browsers already provide host objects that may execute code in unexpected places. Second, most intercepted operations could in general already trigger arbitrary code (for instance, a property access `obj.x` could implicitly invoke an accessor property). Proxies do enable new code paths in the case of the `new`, `in` and `delete` operators and `for-in` loops.

5.3.3. Summary. It is hard to make general claims about what operations can or cannot be intercepted by Proxy APIs. It is a design decision that depends on the particulars of the system at hand. The point that we want to make is that such a tradeoff exists. This brings us to the following principle:

Selective interception: for each operation applicable to objects, consider carefully whether a Proxy API should be able to intercept the operation. For stable operations in particular, consider bypassing the proxy handler to guarantee stability of the result.

5.4. Transparent Virtualization and Handler Encapsulation

An important design point for a Proxy API is whether or not to introduce an operation that can test whether an object is a proxy. For instance, should the API provide a `Reflect.isProxy(obj)` operation that exposes the true nature of `obj`? We decided against introducing such a general test since first, it would directly break transparent virtualization (i.e. the ability to substitute a concrete API with a virtual one without affecting clients) and second, the invariant enforcement mechanism makes it unnecessary for clients to protect themselves against proxies in general, taking away a major incentive to be able to discriminate proxies.

While we consider a *general* `isProxy` test to be a bad idea, an *application-specific* test is often useful. An application may want to test whether an object is a particular kind of proxy, typically generated by the application itself. An application could implement such a test without support from the Proxy API by, e.g. registering all of its created proxies in a set, and then testing whether the object is in the set. The main drawback of this approach is the potential for memory leaks. Luckily, the next edition of ECMAScript will provide just the right abstraction to mitigate this, called a *WeakMap*:

```
function MyProxyFactory() {
  var proxies = new WeakMap();
  return {
    create: function(target, handler) {
      var proxy = Proxy(target, handler);
      proxies.set(proxy, true);
      return proxy;
    },
    isProxy: function(proxy) {
      return proxies.get(proxy) || false;
    }
  };
}
```

A `WeakMap` is like a non-enumerable, object-identity hashtable that weakly references its keys, with the additional property that it avoids a crucial memory leak when cyclic references exist from values stored in the table back to the keys. Its implementation is based on Ephemerons [Hayes 1997]. Essentially, the `WeakMap` enables one to retain an identity-based mapping without introducing memory leaks.

The above pattern captures how an application might introduce a custom proxy factory that supports an application-specific `isProxy` test. This test reliably identifies proxies created by the application's *specific* factory.

We can pose a similar design question for the encapsulation of proxy handlers: is it a good idea to provide a generally applicable `Reflect.getHandlerOf(proxy)` operation that returns the handler of a proxy? Again, we decided against such an operation since it directly breaks handler encapsulation. In our API, given a reference to a proxy, one cannot gain direct access to the proxy's handler. If a handler can be encapsulated behind its proxy, one can ensure that its traps are only ever invoked by manipulating its corresponding proxy, and the trap arguments will always be of a correct type.

Here, again, while a *general* `getHandlerOf` operation is potentially harmful, an *application-specific* variant is often useful. The general technique to accomplish it is the same as for the `isProxy` test shown above: the `create` method can simply register a mapping from `proxy` to `handler`, rather than to `true`. A `getHandlerOf` operation then simply queries the `proxies` WeakMap for the handler. To summarize:

A Proxy API can support **transparent virtualization** if it provides no general mechanism to detect whether an object is a proxy. It can support **handler encapsulation** if it provides no general mechanism to access the handler of a proxy.

6. VIRTUAL OBJECTS AND VIRTUAL HANDLERS

The Proxy API discussed thus far assumes that a proxy always usefully wraps a target object. This need not be the case for all use cases of proxies. A second major use case for proxies is the implementation of *virtual* objects, that is, objects that are not necessarily backed by a non-proxy object in the same address space. A prominent example of such objects in Javascript are the host objects discussed in Section 3.4. Proxies can be used to implement virtual host objects in Javascript itself. Other use cases of virtual objects include proxies for objects persisted in a database and remote object proxies to support distributed programming.

6.1. Virtual Objects

To create a virtual object using our Proxy API, it suffices to create a proxy with a dummy (perhaps empty) target object, and to have the handler ignore that target object. There are a couple of caveats though:

(1) The invariant enforcement mechanism discussed in Section 5.2 will not allow the handler to expose non-configurable properties or emulate a non-extensible object, unless the dummy object stores the exposed non-configurable properties or is itself non-extensible.
(2) To properly emulate the object and fully ignore the target object, the handler object *must* now implement *all* the traps. Recall that when missing, the default implementation of a trap is to forward the intercepted operation to the target object. For virtual objects, this default behavior is undesirable and should be overridden.

6.1.1. VirtualHandler. To mitigate the second issue, our API provides a `VirtualHandler` prototype. Virtual object handlers can inherit from this prototype to acquire a full implementation of the handler API. However, since proxies implementing virtual objects cannot default to forwarding to their target object anymore, the `VirtualHandler` must resort to a different default implementation of the handler API.

As shown in Table I, our handler API defines 15 traps in total, each intercepting a different Javascript operation applied to a proxy. However, some traps intercept an operation that can be expressed in terms of other operations. We refer to such traps as *derived* traps, since their implementation may be derived from other, more *fundamental* traps.

The `VirtualHandler` treats all fundamental traps as *abstract methods*: their default implementation simply throws an exception. The intent is for programmers to override each fundamental trap with a meaningful interpretation for their own virtual object abstraction. The utility of the `VirtualHandler` lies in its implementation of all the derived traps. For each of the 9 derived traps, the `VirtualHandler` implements these derived operations as template methods in terms of the fundamental traps. The net result is that a handler inheriting from a `VirtualHandler` need only implement the 6 fundamental traps, rather than all 15 traps.

For example, the default implementation of the `hasOwn` derived trap is as follows, based on the fact that for a regular object `obj`, a property `name` is an own property of `obj` *if and only if* `Object.getOwnPropertyDescriptor(obj,name)` does not return `undefined`:

```
VirtualHandler.hasOwn = function(target, name) {
  var desc = this.getOwnPropertyDescriptor(target, name);
  return desc !== undefined;
}
```

This default implementation does not depend on the `target` object, instead defining the semantics of `hasOwn` in terms of a call to the `getOwnPropertyDescriptor` method, which may be overridden. The other derived traps are defined in a similar way[6].

When dealing with derived traps, one should consider:

Efficiency. The default implementation of derived traps is often less efficient than a direct implementation. The fundamental traps invoked as part of the implementation of a derived trap perform object allocations that can often be avoided with a more direct implementation. For example, the derived `hasOwn` trap calls `getOwnPropertyDescriptor`, which may allocate a property descriptor, only to subsequently compare it against `undefined`. A virtual object handler is free to override the default derived traps to provide a more efficient implementation.

Consistency. If a derived trap is overridden, care should be taken that its behavior matches that of its dependent fundamental traps. For example, if a virtual object handler overrides both the `hasOwn` and the `getOwnPropertyDescriptor` traps, these traps may return mutually inconsistent results: the `hasOwn` trap could return `true` while `getOwnPropertyDescriptor` could return `undefined`. It is up to the implementor of a virtual object abstraction to define "well-behaved" handlers that uphold the consistency between fundamental and derived traps.

6.2. Proxies as Handlers

A common pattern in proxy handlers is to perform a check to decide whether or not the intercepted operation can be forwarded to the target object. Since there are 15 different operations that proxy handlers can intercept, a straightforward implementation would have to duplicate the pattern of access checking and forwarding in each trap. It is not trivial to make abstraction of this pattern, because each operation has to be intercepted and forwarded differently.

Ideally, if all operations could be uniformly funnelled through a single trap, the handler would only have to perform the access check once, in the single trap. Such funneling of all operations through a single trap can be achieved by implementing the proxy handler itself as a proxy. In meta-programming terms, this corresponds to shifting to the meta-meta-level. The code below demonstrates this:

```
var mh = {
  get: function(dummyTarget, trapName) {
    // code here is run before every operation intercepted by p,
    // e.g. access control checks. To proceed, return a forwarding method
    return Reflect[trapName];
  }
});
var dummy = {};
var bh = Proxy(dummy, mh);
var p = Proxy(t, bh);
```

Figure 2 depicts how meta-level shifting works. The proxy `p` is the proxy object with which other regular application objects directly interact. That proxy's handler is the *base handler* `bh`. The crucial point is that the base handler `bh` is itself a proxy. The handler of `bh` is a *meta handler* `mh`.

Note that *all* operations performed on `p` are intercepted by `mh`'s *single* `get` trap. This pattern works because we carefully designed the API such that proxies *exclusively*

[6]While the `VirtualHandler` can be fully expressed in Javascript itself, we opted to make it part of the standard, to plan for language growth. If a future edition of the language introduces a new derived operation, the `VirtualHandler` can provide a default implementation for it. Proxy code written for a previous edition, using the built-in `VirtualHandler`, will then support the new derived operation without modification.

interact with their associated handler by performing a property access to lookup a trap, and then invoke it if it exists. Proxies never assign traps, enumerate traps or query their handler for the presence of a trap. Because of this uniformity, if the base handler `bh` is *only* used in its role as the handler for a proxy `p`, property access is the only operation that will be performed on `bh`, and the meta handler `mh` only needs to implement the `get` trap.

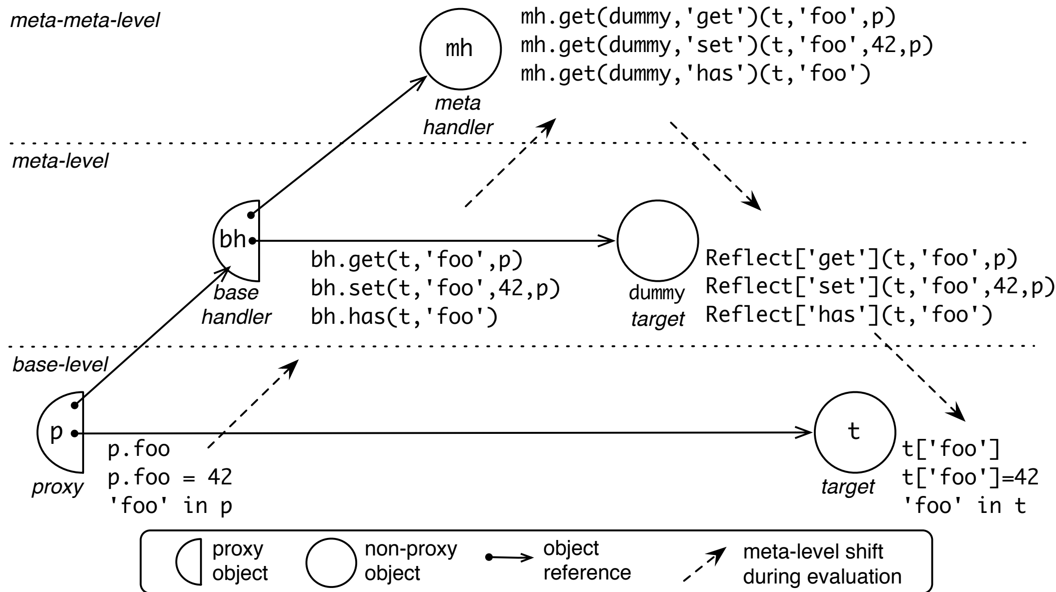

Fig. 2. Meta-level shift using a proxy as a handler

Another way to explain why this funneling works is as follows. At the base-level, programs may perform multiple operations on Javascript objects (e.g. property lookup, assignment, enumeration, . . .). At the meta-level (that is: in the Proxy API), all of these operations are reified uniformly as method invocations on handlers. Therefore, at the meta-level, the only operation performed on meta-level objects (handlers) is property access, and meta-meta-level objects (meta handlers) only need to handle this single operation in their protocol.

If one shifts an extra level upwards to funnel all operations through a single trap of a meta-level handler, one must also shift an extra level downwards if these operations must eventually be applied to a base-level object. This can be accomplished by selecting the appropriate generic forwarding method from the `Reflect` object. The forwarding method then translates the intercepted operation back into a base-level operation on the target object.

Naturally, one can further shift meta-levels at the meta-meta-level. The API allows an arbitrary number of such meta-level shifts. This brings up the question of infinite meta-regress. Such an infinite regress is avoided as long as a handler is eventually implemented as a concrete Javascript object, rather than as yet another proxy.

To summarize, this section reveals two additional design principles of Proxy APIs:

Meta-level shifting: a Proxy API supports meta-level shifting if a proxy handler can itself be a proxy.

Meta-level funneling: the interaction between a proxy and its handler determines the API of the meta-meta-level. If the proxy only uses the handler to lookup traps, then the meta-meta-level API collapses to a single trap through which all meta-level operations are funnelled.

In Section 8 we formally show that meta-level funneling correctly forwards each intercepted operation to the target object. A practical application of meta-level funneling is given in the next Section.

7. ACCESS CONTROL WRAPPERS

Now that our reflection API has been introduced in detail, we can turn to more concrete applications of the API. In this section, we will focus on implementing generic object wrappers that implement a form of access control. The following section describes the implementation of a simple revokable reference wrapper around a single object. Subsequently in Section 7.2 we show how this abstraction can be generalized to support the transitive wrapping of entire object graphs.

7.1. Revokable Object References

Recall the caretaker pattern described in Section 4. This abstraction enabled `Alice` to wrap a precious resource `Carol` in a revokable reference (a proxy), pass that proxy to `Bob`, and hold on to a `revoke()` function that allowed her to later revoke `Bob`'s access to `Carol`. Below is an implementation of this abstraction using our Proxy API:

```
function makeCaretaker(target) {
  var enabled = true;
  return {
    ref: Proxy(target, {
      get: function(...args) {
        if (!enabled) { throw new Error("revoked"); }
        return Reflect.get(...args);
      },
      has: function(...args) {
        if (!enabled) { throw new Error("revoked"); }
        return Reflect.has(...args);
      },
      // ... and so on for all other traps
    }),
    revoke: function() { enabled = false; }
  };
}
```

Note the repetitive pattern in the caretaker proxy's traps. As discussed in Section 6.2, this is a pattern that can itself be abstracted by shifting meta-levels once more. If the caretaker's handler is itself a proxy, the meta-level handler can funnel all meta-level operations through a single `get` trap:

```
function makeCaretaker(target) {
  var enabled = true;
  var baseHandler = Proxy({}, {
    get: function(dummyTarget, trapName) {
      if (!enabled) { throw new Error("revoked"); }
      return Reflect[trapName];
    }
  });
  return {
    ref: Proxy(target, baseHandler),
    revoke: function() { enabled = false; }
  };
}
```

A limitation of the above caretaker abstraction is that objects exchanged via the caretaker are themselves not recursively wrapped in a revocable reference. For example, if `Carol` defines a method that returns `this`, she exposes a direct reference to herself to `Bob`, circumventing `Alice`'s caretaker. This is an instance of the *two-body problem* [Eugster 2006], the fact that wrappers may be "leaky" because the wrapping proxy and the wrapped object are distinct entities. The abstraction discussed in the following section addresses this issue.

7.2. Membranes

A membrane is an extension of a caretaker that transitively imposes revocability on all references exchanged via the membrane [Miller 2006]. One use case of membranes lies in the composition of code from untrusted third parties on a single web page (so-called "mash-ups"). Assuming the code is written in a safe subset of Javascript, such as Caja [Miller et al. 2008], loading the untrusted code inside such a membrane can fully isolate scripts from one another and from their container page. Revoking the membrane around such a script renders it instantly powerless.

The following example demonstrates the transitive effect of a membrane. The prefix `wet` identifies objects initially inside of the membrane, while `dry` identifies revokable references outside of the membrane designating wet objects.

```
var wetA = { x: 1 };
var wetB = { a: wetA };
var membrane = makeMembrane(wetB);
var dryB = membrane.ref; // a proxy for wetB
var dryA = dryB.a;
dryA.x // returns 1, primitives are not wrapped
membrane.revoke(); // revokes all dry references at once
dryB.a // error: revoked
dryA.x // error: revoked
```

The interface of a membrane is the same as that of a caretaker. Its implementation is shown below. A membrane consists of one or more wrappers. Every such wrapper is created by a call to the `wrap` function. All wrappers belonging to the same membrane share a single `enabled` variable. Assigning the variable to `false` instantaneously revokes all of the membrane's wrappers.

The `baseHandler` handler of each wrapper proxy is itself a proxy that implements just the `get` trap (meta-level funneling). This trap returns a function, to be invoked by the wrapper proxy as a trap. If the membrane is not revoked, this function generically performs the intercepted operation on the `target` via the `Reflect[trapName]` method, with appropriate wrapping of arguments and return values that cross the membrane. Arguments cross the membrane in one direction, returned values or thrown exceptions in the other direction.

The wrapper proxy wraps a `dummyTarget` rather than the original `target`. If the target to be wrapped is a function, the `dummyTarget` should itself be a function. Otherwise, as explained in Section 4.2, the `apply` and `construct` traps would not be enabled on the wrapper proxy. If the target to be wrapped is not a function, the `dummyTarget` inherits from a wrapped version of the original `target`'s prototype. Why is this substitution of the real target for a dummy target needed? As explained in Section 5.3.1, to guarantee stability, the built-in function `Object.getPrototypeOf(p)`, when applied to a proxy `p`, reveals the prototype of `p`'s target without consulting `p`'s handler. Had we passed the original `target` as the target of the membrane wrapper proxy, `Object.getPrototypeOf` would leak the unwrapped prototype. Using the substitution, it reveals `dummyTarget`'s prototype, which is a wrapped version of the real target's prototype.

```
function makeMembrane(initTarget) {
  var enabled = true;
  function wrap(target) {
    //primitives provide only irrevocable knowledge, no need to wrap them
    if (isPrimitive(target)) { return target; }
    var baseHandler = Proxy({}, {
      get: function(ignoreTarget, trapName) {
        if (!enabled) {throw new Error("revoked");}
        return function(dummyTarget,...args) {
          try {
            return wrap(Reflect[trapName](target,...args.map(wrap)));
          } catch (e) { throw wrap(e); }
        }
      }
    });

    var dummyTarget = (typeof target === "function") ?
      function() {} :
      Object.create(wrap(Object.getPrototypeOf(target)));
    return Proxy(dummyTarget, baseHandler);
  }
  return {
    ref: wrap(initTarget),
    revoke: function() { enabled = false; }
  };
}
```

The above code represents a minimal but working implementation of the membrane pattern. This minimal implementation has a number of issues that more complete implementations can and should address:

Invariant-preserving Membranes. Since the target of the wrapper proxies is only a surrogate, empty `dummyTarget`, the invariant enforcement mechanism described in Section 5.2 will prevent the membrane from accurately exposing certain invariants. For instance, since the empty dummy target does not itself define non-configurable properties, the membrane proxy cannot expose non-configurable properties of the actual target. If it does, the invariant enforcement mechanism will throw an exception instead, since it cannot guarantee, by simple inspection of the empty dummy target, that the property is indeed non-configurable. This can be amended by having the membrane proxy store non-configurable target properties on the dummy target right before exposing them. In general, the handler has to keep the dummy object "in sync" with the real target object.

Identity-preserving Membranes. Regarding object identity, there are two issues with the above membrane code. First, no distinction is made between the opposite directions in which an object can cross a membrane. Hence, passing a dry object through the membrane in the opposite direction does not turn it into the original wet object. Second, the wrappers are not cached, so if an object is passed through the same membrane twice, clients will receive two distinct wrappers `dry1` and `dry2`, such that `dry1 !== dry2`, even though both denote the same wet object. These limitations can be addressed by having the membrane maintain two WeakMaps.

The first WeakMap maps wet objects to dry wrappers and wet wrappers to dry objects. The second maps dry wrappers to wet objects and dry objects to wet wrappers, respectively. The mapping from objects to wrappers ensures that only one canonical wrapper is created per object. The reverse mapping allows wrappers to be unwrapped when they cross the membrane in the opposite direction. Upon revocation, an identity-

preserving membrane would drop all references to these WeakMaps, making all of their content potentially garbage-collectable. This fact, combined with the garbage-collection properties of WeakMaps, ensures that membranes cannot generate memory leaks, even if there exist cycles among wet and dry objects.

Implementations of both invariant- and identity-preserving membranes are available via our `es-lab.googlecode.com` project.

8. OPERATIONAL SEMANTICS

We now present an extension of the untyped λ-calculus with records, record-inheritance and proxies, named JS-PROXY. Our calculus is inspired by the λ_{proxy} calculus of Austin et al. [Austin et al. 2011], which models proxies representing virtual values (see Section 10.2).

The JS-PROXY calculus models the essential subset of our Javascript proxies API. The language modelled by the calculus includes functions, records, inheritance between records and a `typeof` operator inspired by Javascript's `typeof` operator. Records can have mutable and immutable properties. An immutable property corresponds to a non-configurable, non-writable data property in Javascript. The `isConst r s` operator can be used to test whether the property `s` of a record `r` is immutable.

In the JS-PROXY calculus, proxies can intercept property lookup (`get` trap), property assignment (`set` trap) and function application (`apply` trap). The calculus models the interception of these operations and additionally the interaction between proxies and object-based inheritance (Section 4.3), the invariant enforcement mechanism (Section 5.2) and selective interception of the `typeof` and `isConst` operations (Section 5.3). JS-PROXY supports transparent virtualization and handler encapsulation (Section 5.4). We also revisit the Reflect API (Section 4.4) and meta-level shifting and funneling (Section 6.2) in the JS-PROXY calculus.

Records are represented as tuples (f, r) where f is a partial function from strings to property descriptors representing the record's own properties and r is the object's prototype. A property descriptor in JS-PROXY is simply a tuple (v, b) where v denotes the property value and b is a boolean indicating whether the property is constant (immutable).

The rules [GET], [GETPROTO] and [GETMISSING] together implement record lookup. Record lookup proceeds up the prototype hierarchy until a null prototype is encountered. If the property is not found, null is returned. Record update does not proceed up the prototype hierarchy[7]. Note that the [SET] rule only allows updates to mutable properties. It is an error to update an immutable property. If a record is updated with a non-existent property, the property is added to the record.

The `isConst` operator returns the mutability flag for own properties. Non-existent properties are always reported as mutable[8]. The `typeof` operator returns a string that classifies a value as either a function, a record or a constant.

The rules [FWDAPPLYPROXY], [FWDGETPROXY] and [FWDSETPROXY] implement the default forwarding behavior of proxies: if a trap for the intercepted operation does not exist on the handler object h, the intercepted operation is performed unmodified on the target object t.

The next four rules describe the intercepting behavior of proxies when a proper trap w is defined on the handler. According to rule [APPLYPROXY], when a proxy is applied,

[7] In actual Javascript, property update does climb the prototype chain, to trigger potentially inherited accessor properties.

[8] Note that record update can be used to override an immutable inherited property with a new mutable own property. This is not the case in ECMAScript 5, but remains a topic of ongoing discussion within the committee.

The JS-PROXY calculus

Syntax

e ::=		**Expressions**
	x	variable
	c	constants
	$\lambda x.e$	abstraction
	$e\ e$	application
	$\{\overline{e : e}, \overline{e = e}\} \lhd e$	record creation
	$e[e]$	record lookup
	$e[e] := e$	record update
	proxy $e\ e$	proxy creation
	typeof e	type test
	isConst $e\ e$	mutability test
c ::=	s | null | true | false	**Constants**

Syntactic Sugar

$$e.x \stackrel{\text{def}}{=} e[\texttt{"x"}]$$
$$e.x := e' \stackrel{\text{def}}{=} e[\texttt{"x"}] := e'$$
$$x : e \stackrel{\text{def}}{=} \texttt{"x"} : e$$
$$x = e \stackrel{\text{def}}{=} \texttt{"x"} = e$$
$$\{\overline{e : e}, \overline{e = e}\} \stackrel{\text{def}}{=} \{\overline{e : e}, \overline{e = e}\} \lhd \text{null}$$
$$e\ ;\ e' \stackrel{\text{def}}{=} (\lambda x.e')\ e \quad x \notin FV(e')$$
$$e\ e'\ e'' \stackrel{\text{def}}{=} (e\ e')\ e''$$

its handler's `apply` trap w is applied to the target and the original argument. This rule is only allowed if the proxy wraps a function (either directly or indirectly).

The [GETPROXY] rule is only applicable when intercepting non-constant properties. The rule [CONSTGETPROXY] enforces the invariant that the value of an immutable property cannot change. The value v returned by applying the `get` trap w must be the same as the value v returned by $t[s]$. The rule [SETPROXY] only enables record update for non-constant properties. It also only allows execution to proceed if applying the `set` trap w returns `true`. If the `set` trap returns `false` or another value, evaluation is stuck (in the actual Javascript API, an exception would be thrown instead).

The [ISCONSTPROXY] and [TYPEOFPROXY] rules, finally, show that the proxy handler is unable to intercept these operations. This property enables us to safely depend on the outcome of these operators to enforce the invariants in the conditions of the other rules.

The Reflect API introduced in Section 4.4 can be expressed in the JS-PROXY calculus as follows:

JS-PROXY SEMANTICS

Runtime Syntax

$$
\begin{array}{lll}
h & ::= a \mid \text{proxy}\ t\ h & \text{Records} \\
t & ::= h \mid \lambda x.e & \text{Proxy targets} \\
r & ::= \text{null} \mid h & \text{References} \\
v, w & ::= c \mid t & \text{Values} \\
e & ::= \ldots \mid a & \text{Expressions with addresses} \\
D & ::= \mathit{Value} \times \mathit{Boolean} & \text{Descriptors} \\
H & ::= \mathit{Address} \rightarrow_p (\mathit{String} \rightarrow_p \mathit{Descriptor}) \times \mathit{Reference} & \text{Heaps} \\
E & ::= \bullet\ e \mid v\ \bullet \mid \{\overline{s : v}, s : \bullet, \overline{s : e}, \overline{s = e}\} \triangleleft e & \text{Evaluation context frames} \\
 & \mid\ \{\overline{s : v}, \overline{s = v}, s = \bullet, \overline{s = e}\} \triangleleft e \mid \{\overline{s : v}, \overline{s = v}\} \triangleleft \bullet & \\
 & \mid\ \text{proxy}\ \bullet\ e \mid \text{proxy}\ v\ \bullet \mid \bullet\,[e] \mid v[\bullet] \mid \bullet\,[e] := e & \\
 & \mid\ v[\bullet] := e \mid v[w] := \bullet \mid \text{typeof}\ \bullet & \\
 & \mid\ \text{isConst}\ \bullet\ e \mid \text{isConst}\ v\ \bullet &
\end{array}
$$

Evaluation Rules

$H, \{\overline{s : v}, \overline{s = v}\} \triangleleft r$	$\rightarrow$	$H[a := (f, r)], a$	$f(s) = \begin{cases} (v, \text{false}) & \text{if } s : v \in \overline{s : v} \\ (v, \text{true}) & \text{if } s = v \in \overline{s = v} \end{cases}$ $a \notin \mathit{dom}(H)$	[ALLOC]
$H, (\lambda x.e)\ v$	$\rightarrow$	$H, e[x := v]$		[APPLY]
$H, a[s]$	$\rightarrow$	H, v	$H(a) = (f, r), f(s) = (v, b)$	[GET]
$H, a[s]$	$\rightarrow$	$H, r[s]$	$H(a) = (f, r), s \notin \mathit{dom}(f), r \neq \text{null}$	[GETPROTO]
$H, a[s]$	$\rightarrow$	H, null	$H(a) = (f, \text{null}), s \notin \mathit{dom}(f)$	[GETMISSING]
$H, a[s] := v$	$\rightarrow$	$H[a := (f', r)], v$	$H(a) = (f, r), f' = f[s := (v, \text{true})]$ $f(s) \neq (w, \text{false})$	[SET]
$H, \text{isConst}\ a\ s$	$\rightarrow$	H, b	$H(a) = (f, r), f(s) = (v, b)$	[ISCONST]
$H, \text{isConst}\ a\ s$	$\rightarrow$	H, false	$H(a) = (f, r), s \notin \mathit{dom}(f)$	[ISCONSTFALSE]
$H, \text{typeof}\ (\lambda x.e)$	$\rightarrow$	$H, \texttt{"function"}$		[TYPEOFFUN]
$H, \text{typeof}\ a$	$\rightarrow$	$H, \texttt{"record"}$		[TYPEOFREC]
$H, \text{typeof}\ c$	$\rightarrow$	$H, \texttt{"constant"}$		[TYPEOFCST]
$H, (\text{proxy}\ t\ h)\ v$	$\rightarrow$	$H', t\ v$	if $H, h.\text{apply} \rightarrow^* H', \text{null}$	[FWDAPPLYPROXY]
$H, (\text{proxy}\ t\ h)[s]$	$\rightarrow$	$H', t[s]$	if $H, h.\text{get} \rightarrow^* H', \text{null}$	[FWDGETPROXY]
$H, (\text{proxy}\ t\ h)[s] := v$	$\rightarrow$	$H', t[s] := v$	if $H, h.\text{set} \rightarrow^* H', \text{null}$	[FWDSETPROXY]
$H, (\text{proxy}\ t\ h)\ v$	$\rightarrow$	$H', w\ t\ v$	if $H, h.\text{apply} \rightarrow^* H', w$ and $w \neq \text{null}$ and $H', \text{typeof}\ t \rightarrow^* H', \texttt{"function"}$	[APPLYPROXY]
$H, (\text{proxy}\ t\ h)[s]$	$\rightarrow$	$H', w\ t\ s$	if $H, h.\text{get} \rightarrow^* H', w$ and $w \neq \text{null}$ and $H', \text{isConst}\ t\ s \rightarrow^* H', \text{false}$	[GETPROXY]
$H, (\text{proxy}\ t\ h)[s]$	$\rightarrow$	H''', v	if $H, h.\text{get} \rightarrow^* H', w$ and $w \neq \text{null}$ and $H', \text{isConst}\ t\ s \rightarrow^* H', \text{true}$ and $H', w\ t\ s \rightarrow^* H'', v$ and $H'', t[s] \rightarrow^* H''', v$	[CONSTGETPROXY]
$H, (\text{proxy}\ t\ h)[s] := v$	$\rightarrow$	H'', v	if $H, h.\text{set} \rightarrow^* H', w$ and $w \neq \text{null}$ and $H', \text{isConst}\ t\ s \rightarrow^* H', \text{false}$ and $H', w\ t\ s\ v \rightarrow^* H'', \text{true}$	[SETPROXY]
$H, \text{isConst}\ (\text{proxy}\ t\ h)\ s$	$\rightarrow$	$H, \text{isConst}\ t\ s$		[ISCONSTPROXY]
$H, \text{typeof}\ (\text{proxy}\ t\ h)$	$\rightarrow$	$H, \text{typeof}\ t$		[TYPEOFPROXY]
$H, E[e]$	$\rightarrow$	$H', E[e']$	if $H, e \rightarrow H', e'$	[CONTEXT]

$$\text{Reflect} \stackrel{\text{def}}{=} \begin{array}{rcl} \{\text{apply} & = & \lambda x_t.\lambda x_v.(x_t\ x_v), \\ \text{get} & = & \lambda x_t.\lambda x_s.(x_t[x_s]), \\ \text{set} & = & \lambda x_t.\lambda x_s.\lambda x_v.(x_t[x_s] := x_v; \text{true})\} \end{array}$$

The `Reflect` record has a function-valued property for each of the three proxy traps, and performs the intercepted operation on the target object. We now use this definition of `Reflect` to demonstrate the meta-level shifting and funneling technique introduced in Section 6.2. We do so by showing that all three interceptable operations (function application, record lookup and update), when applied to an identity proxy implemented using meta-level shifting, are equivalent to applying the intercepted operation to the proxy's underlying target. The identity proxy `IdProxy` is defined as follows:

$$\text{IdProxy} \stackrel{\text{def}}{=} \lambda x_t.(\text{proxy}\ x_t\ (\text{proxy}\ \{\}\ \{\text{get} = \lambda x_t.\lambda x_s.(\text{Reflect}[x_s])\ \}))$$

$(\text{IdProxy}\ t)$ is an identity proxy for t. We now show that function application of an identity proxy is equivalent to function application of the underlying target value, i.e. that $H, (\text{IdProxy}\, t)\, v \rightarrow^* t\, v$, assuming function application of t is valid. Thus, assuming $H, \text{typeof}\ t \rightarrow H,$ `"function"`, then:

$$\begin{array}{lll} & H, (\text{IdProxy}\ t)\ v & \\ \rightarrow & H, (\text{proxy}\ t\ (\text{proxy}\ \{\}\ \{\text{get} = \lambda x_t.\lambda x_s.(\text{Reflect}[x_s])\ \}))\ v & \text{IdProxy}, [\text{APPLY}] \\ \rightarrow & H, (\text{proxy}\ \{\}\ \{\text{get} = \lambda x_t.\lambda x_s.(\text{Reflect}[x_s])\ \}).\text{apply}\ t\ v & [\text{APPLYPROXY}] \\ \rightarrow & H, (\{\text{get} = \lambda x_t.\lambda x_s.(\text{Reflect}[x_s])\ \}.\text{get}\ \{\}\ \texttt{"apply"})\ t\ v & [\text{GETPROXY}] \\ \rightarrow & H, ((\lambda x_t.\lambda x_s.(\text{Reflect}[x_s]))\ \{\}\ \texttt{"apply"})\ t\ v & [\text{GET}] \\ \rightarrow & H, (\text{Reflect}[\texttt{"apply"}])\ t\ v & [\text{APPLY}], [\text{APPLY}] \\ \rightarrow & H, (\lambda x_t.\lambda x_v.(x_t\ x_v))\ t\ v & \text{Reflect}, [\text{GET}] \\ \rightarrow & H, t\ v & [\text{APPLY}], [\text{APPLY}] \end{array}$$

Similarly, it can be shown that $H, (\text{IdProxy}\ t)[s] \rightarrow^* H, t[s]$ and $H, (\text{IdProxy}\ t)[s] := v \rightarrow^* H, t[s] := v$ (only if $t[s] := v$ is well-defined, i.e. if $H, \text{isConst}\ t\ s \rightarrow H, \text{false}$). Thus, all operations interceptable by proxies, when applied to $(\text{IdProxy}\ t)$, are equivalent to applying the operation directly to t. This shows that meta-level shifting and funneling behave as expected.

9. PROTOTYPE IMPLEMENTATION

At the time of this writing, our earlier Proxy API for Javascript [Van Cutsem and Miller 2010] is available in Mozilla Firefox 8. It is also available on Google's v8 Javascript engine. The API as discussed here is not yet available. However, we have implemented a Javascript wrapper library that implements the Proxy API as discussed in this paper on top of the older API. Using this library, all of the examples from this paper can be readily executed in Firefox 8. The wrapper library can be downloaded from `es-lab.googlecode.com`.

In previous work we have reported on micro-benchmarks of the existing prototype implementations [Van Cutsem and Miller 2010]. The most important conclusion to be drawn from these micro-benchmarks is that proxies introduce no measurable overhead for regular non-proxy objects. In other words, the additional code required for proxies to intercept does not interfere with the fast path of modern VMs. Naturally, a language operation intercepted by a proxy does introduce overhead. The existing prototype implementations do not yet support built-in invariant enforcement (Section 5.2). We therefore refrain from posting premature micro-benchmark results that could be misleading.

In the particular case of access control wrappers such as caretakers and membranes, proxies are expected to be only introduced between the boundaries of larger subsystems. As long as proxies are not used at a very fine-grained level, the overhead of these abstractions should not be problematic.

10. RELATED WORK

10.1. OO Intercession APIs

In this section we describe a variety of message-based, object-oriented intercession APIs, and briefly compare and contrast them with our Javascript Proxy API. We do not claim that our survey is exhaustive, but we believe the most representative intercession APIs are covered.

Java The Java 1.3 `java.lang.reflect.Proxy` API is a major precedent to our Javascript proxy API. Java's dynamic proxies can be used to intercept invocations on instances of interface types:

```
InvocationHandler h = new InvocationHandler() {
  Object invoke(Object pxy, Method m, Object[] args) {...}
};
Foo proxy = (Foo) Proxy.newProxyInstance(classldr, new Class[] { Foo.class }, h);
proxy.bar(a,b); // triggers h.invoke(proxy, barMethod, new Object[]{a,b})
```

Here, `proxy` implements the `Foo` interface. `h` is an object that implements a single `invoke` method. Method invocations on `proxy` trigger the `h.invoke` method.

The major points of difference between Java proxies and Javascript proxies are first that Java proxies can only intercept a single operation – method invocation. There are no other meta-level operations to trap. For instance, since interfaces cannot declare fields, proxies do not need to intercept field access. Second, Java proxies do not necessarily wrap a target object. Third, Java proxies have no need for the elaborate invariant enforcement mechanism required in Javascript proxies. The only invariant to be maintained by Java proxies is that the runtime type of the return value of the `invoke` method must be compatible with the static return type of the intercepted method.

The Java Proxy API can only construct proxies for interface types, not class types. As a result, proxies cannot be used in any situation where code is typed using class types rather than interface types, limiting their general applicability. Eugster [Eugster 2006] describes how to extend the Java Proxy API to work uniformly with instances of non-interface classes. Next to the usual `InvocationHandler`, proxies for class types have an additional `AccessHandler` to trap field access.

AmbientTalk The design of Javascript proxies was influenced by AmbientTalk *mirages* [Mostinckx et al. 2007]. AmbientTalk is a distributed dynamic language, with a mirror-based reflection API. AmbientTalk enables intercession through mirages, which are proxy-like objects controlled explicitly by a separate mirror object:

```
def mirage := object: {...} mirroredBy: (mirror: {
  def invoke(receiver, message) { ... };
  def addSlot(slot) { ... };
  def removeSlot(slot) { ... };
  ...
});
```

The mirror is to the mirage what the proxy handler is to a Javascript proxy. Like Javascript proxy handlers, mirrors define an extensive set of traps, enabling near-complete control over the mirage. Unlike our Proxy API, a mirage need not wrap an existing target object.

E is a secure, distributed dynamic language [Miller et al. 2005]. In E, every value is an object, but there are two kinds of object references: *near* and *eventual* refer-

ences. Similarly, there are two message passing operators: immediate call (`o.m()`, a synchronous method invocation) and eventual send (`o<-m()`, an asynchronous message send). Both operations are allowed on near references, but eventual references carry only asynchronous messages. Because of this distinction, E has two separate intercession APIs: one for objects and one for references.

E has a proxy-based API to represent user-defined eventual references [Miller and Reid 2009]:

```
def handler {
  to handleSend(verb :String, args :List) {...}
  to handleSendOnly(verb :String, args :List) {...}
  to handleOptSealedDispatch(brand) {...}
}
def proxy := makeProxy(handler, slot, state);
```

This API is very similar to the one for Javascript proxies. The proxy represents an eventual reference, and any asynchronous send `proxy<-m()` either triggers the handler's `handleSend` or `handleSendOnly` trap, depending on whether the sender expects a return value.

The `handleOptSealedDispatch` trap is part of E's trademarking system and is beyond the scope of this paper. The `slot` argument to `makeProxy` can be used to turn the proxy reference into a *resolved* reference. Once a reference is resolved, the proxy is bypassed and the handler no longer consulted. The `state` argument to `makeProxy` determines the state of the reference. The details are outside the scope of this paper, but by passing this parameter to the `makeProxy` function at construction time, the eventual reference proxy can determine its state without having to consult the handler, thus guaranteeing that the state of a reference remains stable.

E has a distinct intercession API for objects. A *non-methodical object* is an empty object with no methods. Instead, its implementation consists of a single `match` clause that encodes an explicit message dispatch:

```
def obj match [verb, args] {
  # handle the message generically
}
```

The variable `obj` is bound to a new object whose dispatch logic, if any, is explicitly encoded in the `match` clause. An immediate call `obj.m(x)` will trigger this clause, binding `verb` to `"m"` and `args` to a list `[x]`.

Finally, it is worth noting that AmbientTalk inherits from E the distinction between near and eventual references and the distinction between immediate call and eventual send. Unlike E, AmbientTalk has only one intercession API (mirages), but a mirage can represent *both* objects and eventual references, depending on how the handler implements its traps (immediate calls and eventual sends each trigger a separate trap).

Smalltalk Smalltalk-80 popularized generic message dispatch via its `doesNotUnderstand:` mechanism. Briefly, if standard method lookup does not find a method corresponding to a message, the Smalltalk VM instead sends the message `doesNotUnderstand: msg` to the original receiver object. Here, `msg` is an object containing the message's selector and arguments. The default behavior of this method, inherited from `Object` is to throw an exception.

The `doesNotUnderstand:` trap is not stratified. It occupies the same namespace as application-level methods. This lack of stratification did lead Smalltalk programmers to look for alternative interception mechanisms. Foote and Johnson describe a particular extension to ParcPlace Smalltalk called a *dispatching class*: "Whenever an object belonging to a class designated as a dispatching class (using a bit in the class object's header) is sent a message, that object is instead sent `dispatchMessage:`

aMessage." [Foote and Johnson 1989]. Instances of dispatching classes are effectively proxies, the `dispatchMessage:` method acting as the sole trap of an implicit handler.

Ducasse [Ducasse 1999] gives an overview of the various message passing control techniques in Smalltalk. He concludes that `doesNotUnderstand:` is not always the most appropriate mechanism. Rather, he stresses the usefulness of *method wrappers*. This approach was elaborated by Brant *et. al* [Brant et al. 1998]. In this approach, rather than changing the method *lookup*, the method *objects* returned by the lookup algorithm are modified. This is possible because Smalltalk methods and method dictionaries are accessible from within the language. The method wrapper approach is in many ways similar to CLOS method combinations, enabling before/after/around augmentation of existing methods. As their name suggests, they are great for wrapping existing methods, but they seem less suitable to implement virtual objects and thus only support part of the use cases covered by `doesNotUnderstand:`.

Summary We refer to prior work [Van Cutsem and Miller 2010] for a survey on how the OO intercession APIs discussed in this section relate to the design principles put forward in Section 5.

10.2. Virtual Values

Eugster [Eugster 2006] introduced the term *uniform proxies* to denote an object model in which objects of all types can be proxified. Our Proxy API does not in itself support uniform proxies, since one can only proxify objects (including functions and arrays), while Javascript additionally has primitive values (numbers, strings, . . .).

Starting from our initial Proxy API [Van Cutsem and Miller 2010], Austin et al. have recently introduced a complementary Proxy API for virtualizing primitive *values* [Austin et al. 2011]. They focus on creating proxies for objects normally thought of as primitive values, such as numbers and strings. They highlight various use cases, such as new numeric types, delayed evaluation, taint tracking, contracts, revokable membranes and units of measure.

Like our proxies, virtual values are proxies with a separate handler. The handler for a virtual value provides a different set of traps. A virtual value can intercept unary and binary operators applied to it, being used as a condition in an `if`-test, record access and update, and being used as a record index. An important difference between our API and the virtual values API is that the latter does provide a general `isProxy` primitive, deliberately breaking transparent virtualization. The reason is to enable programs to be able to defend themselves against malicious virtual values, such as mutable strings in a language that otherwise only has immutable strings. Virtual values have no invariant enforcement mechanism.

10.3. Chaperones and Impersonators

Chaperones and impersonators are a recent addition to Racket [Strickland et al. 2012]. Chaperones and impersonators are both kinds of proxies. The important difference is that chaperones can only further *constrain* the behavior of the object that it wraps. When a chaperone intercepts an operation, it must either raise an exception, return the same result that the wrapped target object would return, or return a chaperone of the target object's result. Impersonators, on the other hand, are free to change the value returned from intercepted operations.

In Racket, impersonators can only wrap *mutable* data types, while chaperones can wrap both mutable and immutable data types. Chaperones and impersonators form the run-time support for Racket's contract system on higher-order, stateful values.

There is an interesting correspondence between chaperones and impersonators and our Proxy API. The kind of checks that a chaperone must perform are similar to the invariant enforcement mechanism outlined in Section 5.2. Because of this invariant

enforcement mechanism, a Javascript proxy that wraps a frozen object (that is, a non-extensible object with only non-configurable properties) is constrained like a chaperone[9]. Conversely, as long as the wrapped object is fully mutable (i.e. extensible without any non-configurable properties), our Javascript proxies are like impersonators and may modify the result of operations.

10.4. Mirrors

This work is related to Bracha and Ungar's work on mirrors, which revolves around similar design principles for meta-level architectures [Bracha and Ungar 2004]. The principles of stratification and handler encapsulation as stated in this paper are related to the corresponding principles for mirror-based architectures, but with a focus on how they apply to intercession rather than to introspection.

Mirror-based architectures strive to decouple base-level from meta-level code. Traditional reflection APIs usually define access to the reflective interface of an object as part of that object's own interface. A prominent example is the `getClass()` method defined on `java.lang.Object`. The result is a tight coupling between the base-level object and its meta-level representation (in the case of Java the resulting `Class` object). Just like mirrors form a stratified *introspection* API as compared to Java's unstratified `getClass()` method, proxies as discussed here form a stratified *intercession* API as compared to Javascript's unstratified `__noSuchMethod__` trap.

Most mirror-based architectures support introspection and self-modification, but they have limited support for intercession. To the best of our knowledge, AmbientTalk's meta-level architecture based on mirages (Section 10.1) was the first to reconcile mirrors with support for intercession.

10.5. Partial Behavioral Reflection

Partial Behavioral Reflection (PBH) [Tanter et al. 2003] is a framework that describes the spatiotemporal extent of reification. Reflex is an intercession API for Java, based on bytecode rewriting, that supports PBH. Reflex enables the definition of meta-objects for Java objects. A single meta-object can control multiple base-level objects. The spatial scope of meta-objects is delimited using three concepts: *entity selection* enables a meta-object to specify what objects it will control (e.g. all instances of a class, or only a particular instance of a class). *Operation selection* determines what particular operations of the affected objects are reified (e.g. only field access). *Intra-operation selection* enables reification to occur only if the operation satisfies further conditions (e.g. only reify calls to the `foo` method). Finally, *temporal selection* controls the time during which a meta-object is active.

Reflex differs from our Proxy API in that it enables the creation of meta-objects that can act upon objects not explicitly declared as proxies. Nevertheless, some aspects of our proxy API can be understood in terms of PBH. Proxies induce a static form of entity selection: operations on proxy objects are reified, operations on non-proxy objects are not. Proxies may support temporal selection. For example, our earlier Proxy API included a mechanism by which proxies could be switched off, such that they no longer reified operations [Van Cutsem and Miller 2010]. Finally, proxies enable a static form of operation selection: some operations on proxies (e.g. `typeof`) are never reified, whereas others such as property access are always reified.

One could characterize Reflex as a meta-intercession API: using Reflex, one can define many different specific intercession APIs, each with its own settings for entity, operation and temporal selection.

[9]It is actually more constrained, since a chaperone is allowed to return a chaperone for the original value, while our Javascript proxies are not allowed to return a proxy for the value of a non-configurable property.

10.6. CLOS

The Common Lisp Object System (CLOS) has an extensive MOP [Kiczales et al. 1991]. Because CLOS is function-oriented as opposed to a message-oriented, it is difficult to transpose the design principles described in this paper to CLOS. In CLOS, computation proceeds mainly through generic function invocation, as opposed to sending messages to objects. The dispatch mechanism of generic functions can be modified via the MOP. However, if one simply wants to wrap existing methods, CLOS offers a method combination protocol that can be used to insert behavior before, after or around existing methods without modifying the protocol.

11. CONCLUSION

We have reported on the design of the new reflection API of the upcoming ECMAScript standard. Its most prominent feature is the ability to create proxies that are able to intercept a variety of base-level language operations. Proxies are a fundamental new building block that enable a variety of use cases, from generic wrapper abstractions such as membranes and higher-order contracts on mutable objects, to virtual object abstractions and the virtualization of entire Javascript host environment APIs. Compared to existing solutions, the Proxy API is properly stratified, with a clean separation between base-level proxy objects and meta-level handler objects.

In developing this Proxy API, great care has been taken to uphold the non-configurability and non-extensibility invariants of ECMAScript 5. Via a mechanism we call *invariant enforcement*, proxies are allowed to intercept invariant-sensitive operations, but without being able to subvert the invariants. This ensures that proxies cannot become an instrument of attackers to confuse code relying on such invariants.

While our design here focussed mostly on the particulars of ECMAScript, we did call out a number of design principles that are applicable to proxy-based intercession APIs in general. In summary, these are:

Stratification. Traps are defined on a handler object separate from the proxy.
Invariant enforcement. Invariant-sensitive operations can be safely intercepted by proxies if the invariants can be enforced through run-time checks.
Selective interception. To uphold the properties of some operations, they are better not trapped by proxies altogether.
Transparent virtualization. By default, proxies can't be distinguished from regular objects.
Handler encapsulation. By default, a proxy encapsulates its handler.
Meta-level shifting. A proxy handler can itself be a proxy.
Meta-level funneling. Proxies interact with their handlers via a restricted set of operations (ideally just one).

Our hope is that the identification of these principles may help steer the design of similar reflection APIs for other languages.

Acknowledgements

We thank the members of the ECMA TC-39 committee and the `es-discuss@mozilla.org` community for their feedback on our Proxy API. Thanks in particular to Brendan Eich, Andreas Gal, Dave Herman, Sam Tobin-Hochstadt, David Bruant, Andreas Rossberg and Cormac Flanagan for their detailed feedback over the past two years.

REFERENCES

AUSTIN, T. H., DISNEY, T., AND FLANAGAN, C. 2011. Virtual values for language extension. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. OOPSLA '11. ACM, New York, NY, USA, 921–938.

BRACHA, G. AND UNGAR, D. 2004. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA '04: Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications*. 331–343.

BRANT, J., FOOTE, B., JOHNSON, R. E., AND ROBERTS, D. 1998. Wrappers to the rescue. In *ECOOP '98: Proceedings of the 12th European Conf. on Object-Oriented Programming*. Springer-Verlag, 396–417.

CROCKFORD, D. 2008. *Javascript: The Good Parts*. O'Reilly.

DUCASSE, S. 1999. Evaluating message passing control techniques in smalltalk. *Journal of Object-Oriented Programming (JOOP) 12*, 39–44.

ECMA INTERNATIONAL. 2009. *ECMA-262: ECMAScript Language Specification* Fifth Ed. ECMA, Geneva, Switzerland.

EUGSTER, P. 2006. Uniform proxies for java. In *OOPSLA '06: Proceedings of the 21st annual conference on Object-oriented programming systems, languages, and applications*. ACM, NY, USA, 139–152.

FINDLER, R. B. AND FELLEISEN, M. 2002. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. ICFP '02. ACM, New York, NY, USA, 48–59.

FOOTE, B. AND JOHNSON, R. E. 1989. Reflective facilities in smalltalk-80. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*. ACM, 327–335.

GAL, A. AND FLANAGAN, D. 2011. dom.js: Self-hosted javascript implementation of a WebIDL-compliant HTML5 DOM. `https://github.com/andreasgal/dom.js`, accessed December 29, 2011.

HAYES, B. 1997. Ephemerons: a new finalization mechanism. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '97. ACM, New York, NY, USA, 176–183.

KICZALES, G. AND HILSDALE, E. 2001. Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes 26*, 313–.

KICZALES, G., RIVIERES, J. D., AND BOBROW, D. G. 1991. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA.

MILLER, M., TRIBBLE, E. D., AND SHAPIRO, J. 2005. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*. Springer, 195–229.

MILLER, M. S. 2006. Robust composition: Towards a unified approach to access control and concurrency control. Ph.D. thesis, John Hopkins University, Baltimore, Maryland, USA.

MILLER, M. S. AND REID, K. 2009. Proxies for eventual references in E. `wiki.erights.org/wiki/Proxy`.

MILLER, M. S., SAMUEL, M., LAURIE, B., AWAD, I., AND STAY, M. 2008. Caja: Safe active content in sanitized javascript. `tinyurl.com/caja-spec`.

MOSTINCKX, S., VAN CUTSEM, T., TIMBERMONT, S., AND TANTER, E. 2007. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings of the Dynamic Languages Symposium - OOPSLA'07 Companion*. ACM Press, 222–248.

PAEPCKE, A. 1993. *User-level language crafting: introducing the CLOS metaobject protocol*. MIT Press, Cambridge, MA, USA, 65–99.

REDELL, D. D. 1974. Naming and protection in extensible operating systems. Ph.D. thesis, Department of Computer Science, University of California at Berkeley.

STRICKLAND, T. S., TOBIN-HOCHSTADT, S., FINDLER, R. B., AND FLATT, M. 2012. Chaperones and impersonators: Run-time support for contracts on higher-order, stateful values. NU-CCIS-12-01.

TANTER, É., NOYÉ, J., CAROMEL, D., AND COINTE, P. 2003. Partial behavioral reflection: spatial and temporal selection of reification. In *OOPSLA '03: Proceedings of the 2003 Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM, 27–46.

VAN CUTSEM, T. AND MILLER, M. S. 2010. Proxies: design principles for robust object-oriented intercession apis. In *Proceedings of the 6th symposium on Dynamic languages*. DLS '10. ACM, 59–72.