Proxies: Design Principles for Robust Object-oriented Intercession APIs

Tom Van Cutsem *
Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium
tvcutsem@vub.ac.be

Mark S. Miller
Google Research
1600 Amphitheatre Parkway
Mountain View, CA, USA
erights@google.com

Abstract

Proxies are a powerful approach to implement meta-objects in object-oriented languages without having to resort to metacircular interpretation. We introduce such a meta-level API based on proxies for Javascript. We simultaneously introduce a set of design principles that characterize such APIs in general, and compare similar APIs of other languages in terms of these principles. We highlight how principled proxy-based APIs improve code robustness by avoiding interference between base and meta-level code that occur in more common reflective intercession mechanisms.

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

General Terms Design, Languages

Keywords Proxies, Javascript, Reflection, Intercession

1. Introduction

We introduce a new meta-level API for Javascript based on dynamic proxies. Proxies have a wide array of use cases [8]. Two general cases can be distinguished depending on whether the proxy is proxying another object within the same address space:

Generic wrappers are proxies that wrap other objects in the same address space. Example uses include access control, profiling, adaptors that intermediate between different versions of an interface, etc.

Virtual objects are 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 results not yet computed), etc.

The contributions of this paper are first, the introduction of a proxy-based metaprogramming API for Javascript (Section 4), second, the enumeration of a set of design principles that characterize general message-based object-oriented metaprogramming APIs (Sections 4.2 through 4.9), third, a characterization of related metaprogramming APIs in terms of these design principles (Section 7.1) and fourth, the characterization of Proxy APIs as stratified APIs that avoid interference between message interception and application-level code (Sections 3.3 and 4.3).

2. Reflection Terminology

A metaobject protocol (MOP) [11] 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* [11] 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 meta-level APIs. Self-modification is more exceptional, and meta-level APIs with extensive support for intercession

* Tom Van Cutsem is a Postdoctoral Fellow of the Research Foundation, Flanders (FWO). This work was carried out while on a Visiting Faculty appointment at Google, sponsored by Google and a travel grant from the FWO.

DLS 2010, October 18, 2010, Reno/Tahoe, Nevada, USA.

are rare in mainstream languages, the CLOS MOP being a notable exception (see Section 7.4).

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" [5].

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 [7]. 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 [7]. 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). The following code snippet shows how these attributes can be inspected and defined:

```
var pd = Object.getOwnPropertyDescriptor(o, 'x');
// pd = {
//   value: 5,
//   writable: true,
//   enumerable: true,
//   configurable: true
// }
Object.defineProperty(o, '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 is a prototype-based language with delegation). 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*. A non-extensible object cannot be extended with new properties. A sealed object is a non-extensible object whose own (non-inherited) properties are all non-configurable. Finally, a frozen object is a sealed object whose own properties are all non-writable. The call `Object.freeze(obj)` freezes the object `obj`. Section 4.4 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 mapping 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

Javascript has no standard 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 7.1). 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(o) {
  return {
    __noSuchMethod__: function(prop, args) {
      return o[prop].apply(o,args);
    }
  };
}
```

In what follows, 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 purposely 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 may 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. What 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" would quickly grow out of control.

We have discussed how the lack of stratification leads to confusion when a trap is invoked by the wrong caller. There are other ways in which the presence of a trap as part of an object's interface may cause confusion. For example, a common Javascript idiom is to enumerate all properties of an object. Say we want to populate the content of a drop-down list in an HTML form with id id with all of the property names of an object obj:

```
function populateList(id, obj) {
  var select = document.getElementById(id);
  for (var name in obj) {
    // skip inherited properties
    if (!obj.hasOwnProperty(name)) continue;
    var opt = document.createElement('option');
    opt.text = name;
    select.add(opt, null); // append opt
  }
}
```

If obj is a proxy that defines __noSuchMethod__ for the sake of intercepting property access, this method will show up in the enumeration of obj's properties, which is probably not what the code expects.

4. Javascript Proxies

We now describe an intercession API for Javascript. This API is officially proposed for inclusion in the next version of the ECMAScript standard[1].

Our intercession API for Javascript 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.create(handler, proto);
```

Here, handler is an object that must implement a particular meta-level API and proto is an optional argument object representing the proxy's prototype.

Table 1 lists those base-level operations applicable to objects that can be trapped by handlers. The name proxy refers to a proxy instance, handler to that proxy's handler, proto to the prototype specified in Proxy.create and receiver to either a proxy or an object that inherits (directly or indirectly) from a proxy. The distinction between fundamental and derived traps is explained in Section 4.7.

The relationship between traps and the operations they trap should be clear in most cases. The fix trap is described in Section 4.4. 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 iterating over this array. Because Javascript methods are just functions, method invocation, as in proxy.m(a,b) is reified as a property access handler.get(proxy,'m') that is expected to return a function. That function is immediately applied to the arguments [a,b] and with this bound to proxy.

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 system) do not pay the runtime costs associated with intercession (cf. Section 6). Finally, note that the link between a proxy and its handler is immutable and inaccessible to clients of the proxy.

[1] Its draft specification is available at www.tinyurl.com/harmony-proxies.

Operation	Triggered by	Reified as
Fundamental traps		
Own descriptor access	`Object.getOwnPropertyDescriptor(proxy,name)`	`handler.getOwnPropertyDescriptor(name)`
Descriptor access	`Object.getPropertyDescriptor(proxy,name)`	`handler.getPropertyDescriptor(name)`
Descriptor definition	`Object.defineProperty(proxy,name,pd)`	`handler.defineProperty(name,pd)`
Own keys	`Object.getOwnPropertyNames(proxy)`	`handler.getOwnPropertyNames()`
Property deletion	`delete proxy.name`	`handler.delete(name)`
Key enumeration	`for (name in proxy) {...}`	`handler.enumerate()`
Object fixing	`Object.{freeze,seal,preventExtensions}(proxy)`	`handler.fix()`
Derived traps		
Property lookup	`name in proxy`	`handler.has(name)`
Own Property lookup	`({}).hasOwnProperty.call(proxy,name)`	`handler.hasOwn(name)`
Property access	`receiver.name`	`handler.get(receiver, name)`
Property assignment	`receiver.name = val`	`handler.set(receiver, name, val)`
Own enumerable keys	`Object.keys(proxy)`	`handler.keys()`

Table 1. Operations reified on handlers by the Proxy API

4.1 Function Proxies

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: they can be called and constructed. For reasons that will be made clear in Section 4.6, our API explicitly distinguishes between object proxies and function proxies. The call `Proxy.create` returns object proxies. To create a function proxy, one invokes the method `Proxy.createFunction`:

```
var call = function() { ... };
var cons = function() { ... };
var f = Proxy.createFunction(handler, call, cons);
f(1,2,3);  // calls call(1,2,3)
new f(1,2,3);  // calls cons(1,2,3)
f.x  // calls handler.get(f,'x')
```

The first argument to `Proxy.createFunction` is exactly the same kind of handler object passed to `Proxy.create`. It intercepts all operations applicable to functions used as objects. The additional arguments `call` and `construct` are functions that respectively trap Javascript's function call and `new` operator.

Unlike `Proxy.create`, `Proxy.createFunction` does not accept a prototype argument. Functions always inherit from `Function.prototype`, and so do function proxies.

4.2 Generic Wrappers

To implement generic wrappers around existing objects, a useful handler is one that simply forwards all operations applied to the proxy to the wrapped object:

```
function makeForwardingHandler(target) {
  return {
    get: function(rcvr,name) { return target[name];},
    set: function(rcvr,name,val) { target[name] = val; return true; },
    has: function(name) { return name in target; },
    delete: function(name) { return delete target[name]; },
    ...
  };
};
var proxy = Proxy.create(makeForwardingHandler(target),
                         Object.getPrototypeOf(target));
```

A particular generic wrapper can inherit from this forwarder and override only the methods for the operations that it needs to intercept. We make use of this handler in Sections 4.5 and 5.

4.3 Stratification

Bracha and Ungar [2] introduce the principle of stratification for mirror-based architectures. The principle states that meta-level facilities must 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 7.2.

The distinction between a proxy and its handler object enforces stratification of the traps. Traps are not defined as part of the interface of the proxy object, but as part of the interface of the handler.

The handler is a regular object. It may inherit from other objects, and its inheritance chain is completely independent from that of the proxy it handles. A single handler may handle multiple proxies. The handler can be a proxy itself (we will illustrate a use case of this in Section 4.5).

Accessing `aProxy.has` explicitly on a proxy will not trigger the proxy's corresponding `has` trap. Instead, the access will be reified like any other as `handler.get(aProxy,'has')`. Likewise, `aProxy.get` is reified as `handler.get(aProxy,'get')`. Traps can only be invoked explicitly 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`, etc.

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

Stratification: a proxy's traps should be stratified by defining them on a separate handler object.

4.4 Temporary Intercession

Recall from section 3.2 that ECMAScript 5 enables the creation of tamper-proof objects. A tamper-proof object provide useful guarantees that programmers can rely upon. When designing a proxy API, care should be taken that proxies cannot break these guarantees. For example, if `o` is an object, then `Object.freeze(o)` freezes that object. If the programmer knows `o` is frozen, he can rely on the fact that the number of properties contained in `o` will no longer change. If `o` is a proxy, we do not want a proxy to violate such assumptions.

To reconcile non-extensible, sealed and frozen objects with proxies, we introduce an additional trap named `fix`. A call to `Object.freeze(proxy)` will be interpreted as:

```
var props = handler.fix();
if (props === undefined) {
  throw new TypeError();
} else {
  become(proxy, Object.freeze(Object.create(proto, props)));
}
```

Again, `handler` denotes `proxy`'s handler and `proto` denotes the prototype argument passed to the `Proxy.create` call that created the proxy.

The `fix` trap is introduced to enable a proxy to interact with the `Object.preventExtensions`, `Object.seal` and `Object.freeze` primitives available in ECMAScript 5. A non-extensible, sealed or frozen object should restrict the handler's freedom in terms of what it can return from subsequent calls to `has`, `get`, etc. For example, once proxy `p` is frozen, all invocations of `'foo' in p` must return the same value. If this operation traps to the proxy's handler's `has` trap (i.e., as `handler.has('foo')`), then this restriction would be difficult to enforce.

Proxies enforce these restrictions as follows: when an attempt is made to make a proxy non-extensible, sealed or frozen, the `fix` trap is invoked on the proxy's handler. The handler has two options. It can refuse the request (by making its `fix` trap return `undefined`). This causes the operation to throw an exception. Or, the handler can allow the request by returning a "fixed" representation of the proxy's properties. The implementation instantiates this description into an actual object. From that point on the proxy effectively *becomes* that object and the handler is bypassed entirely. The proxy is now said to be "fixed".

The `become` operation used in the above code snippet is pseudo-code: it can only be implemented in the VM. The `props` object returned by the `fix` trap is the same kind of object that can be passed as the second argument to `Object.create`: a record that maps property names to property descriptors.

Because of the fixing-protocol, proxies can essentially be in two states. A proxy is created in a *trapping* state, during which its handler traps all operations applied to the proxy. Once the `fix` trap returns an object description, the proxy is *fixed*. Once a proxy is fixed, it remains forever fixed. A fixed proxy is essentially a "static" proxy, and as far as the Javascript programmer is concerned, it is at that point indistinguishable from a normal Javascript object.

When an object proxy is fixed, the prototype of the generated object is the `proto` object passed as an argument to `Proxy.create`. The prototype of a function proxy is always `Function.prototype`, and this remains unchanged when fixing the function proxy.

The essence of the fixing mechanism can be summed up in the following principle:

Temporary Intercession: if an operation imposes strong consistency requirements on a proxy object (such as becoming immutable), intercession should be disabled for some or all of the operations intercepted by proxies.

Fixing does not necessarily need to be an "all or nothing" process. However, in the particular case of ECMAScript, we found this to be the simplest solution.

4.5 Proxies as Handlers

A common use case for proxy handlers is, for example, to perform an access control check before performing the operation on some wrapped object. Only if the check succeeds is the operation forwarded. Since there are 12 different operations that proxy handlers can intercept, a straightforward implementation would have to duplicate the pattern of access checking and forwarding the operation in each trap. It's difficult to abstract this pattern, because each operation has to be "forwarded" in a different way, as exemplified by the forwarding handler in Section 4.2. Ideally, if all operations could be uniformly funneled 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. Figure 1 depicts how meta-level shifting works. The *base-level proxy* is the proxy object with which other regular application objects directly interact. That proxy's handler is the *base handler*. The crucial part is that the base handler is itself a proxy, call it a *meta-level proxy*. The handler of a meta-level proxy is a *meta handler*.

Note how all operations performed on `p` are intercepted by `mh`'s `get` trap. We carefully designed the intercession API such that proxies only interact with their associated handler by invoking their traps. Proxies never assign, enumerate or test for the presence of traps on their handler. Because of this uniformity, if the base handler is *only* used in its role as the handler for a base-level proxy, the meta handler only needs to implement the `get` trap. Because method invocation is the only operation performed on meta-level proxies, only this single trap is triggered on meta handlers.

Another way to explain why this funneling works is as follows. At the base-level, running programs perform an assortment of operations on Javascript objects (e.g. property

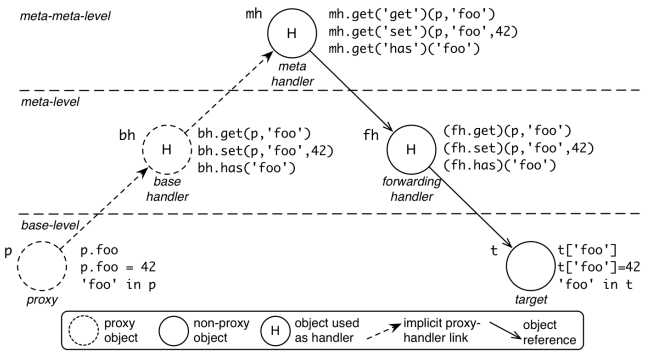

Figure 1. Meta-level shift using a proxy as a handler

lookup, access, assignment, enumeration, ...). At the meta-level (that is: in the intercession 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 (base-level handlers) is property access, and meta-meta-level objects (meta-level 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 having the meta handler forward the operations to a generic forwarding handler. The forwarding handler essentially translates meta-level operations back into base-level operations on a target object.

Naturally, one can apply this trick of shifting meta-levels at the meta-meta-level recursively. 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 important design principles of proxy-based intercession APIs:

Meta-level shifting: a proxy-based intercession API supports shifting to a new meta-level by implementing a handler as 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 invokes the handler's traps, then the meta-meta-level API collapses to a single trap through which all meta-level operations are funneled.

Meta-level funneling is demonstrated in Section 5.1.

4.6 Selective Interception

A proxy-based intercession API introduces a tradeoff 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. We already briefly touched upon this issue in Section 4.4 when discussing how proxies interact with operations that make objects non-extensible, sealed or frozen. As we have discussed, the `fix` trap enables a proxy handler to dynamically decide whether or not to commit to a fixed representation.

Some operations may be so critical that we don't want proxies to ever influence their outcome. One such operation is testing for equality. In Javascript, the expression `a === b` determines whether `a` and `b` refer to identical objects. The `===` operator comes with a number of guarantees that are implicitly taken for granted by programmers. For example, `===` on objects is commutative, transitive, symmetric and deterministic (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, we decided not to enable proxy handlers to trap `===`. Proxies cannot influence the outcome of this operation.

As in the case of `===`, the fact that an operation should be deterministic is a requirement for many operations. To uphold determinism, the operation should not trap to a proxy handler. However, just because an operation should have a deterministic result does not mean that it cannot be reliably intercepted. When an operation requires a deterministic, stable result, this result can be provided once to the proxy when it is created. There are two concrete examples of this in our intercession API.

The first example is that proxies must commit to their prototype upfront. Recall from section 4 that the second argument to `Proxy.create` denotes the proxy's prototype. This has two important effects: the first is that ECMAScript's built-in operation `Object.getPrototypeOf` remains deterministic, even when applied to a proxy. The second is that proxies cannot influence `instanceof` when they are used as the left-hand operand: to determine whether `proxy instanceof Foo` is true, the implementation need only look at the prototype of `proxy`, which using our API it can do without consulting the proxy handler.

The second example is that proxies have to decide upfront whether they will proxy a plain object or a function object. By requiring proxies to commit to representing either an object or a function at proxy creation time, the language runtime is able to uphold important invariants that it would have to forfeit if it would have to consult the `handler`. For example the expression `typeof o` should evaluate to `"function"` if `o` is a function, and to `"object"` otherwise. Also, functions always inherit from `Function.prototype`. The distinction between object and function proxies allows our API to enforce these invariants.

Another aspect to take into account when deciding whether or not a certain operation can be intercepted by a proxy is the fact that it may lead to arbitrary code being run in places where the language runtime or the programmer did not expect this to. For example, without our intercession API, the expression `name in obj` does not trigger Javascript code. Since `in` is a primitive operation, the evaluation of this expression occurs entirely within the language runtime,

without it ever having to run non-native code. Proxies may change these assumptions.

In our particular case, proxies enable non-native code to run while evaluating the `in` operator, the `delete` operator and `for-in` loops, yet they preserve the integrity of `===` and `typeof`. It's hard to make general claims about what operations should generally be interceptable and which should not. It is a design decision that depends on the particulars of the system at hand. The point that we want to make is that it's important to recognize that such a tradeoff exists. This brings us to the following design principle:

Selective interception: determine what properties of an object should be to committed to upfront. If these properties are required to be stable but may be configurable, turn them into parameters of the proxy constructor rather than retrieving them via a handler trap.

4.7 Fundamental versus derived traps

As shown in Table 1, our handler API defines 12 traps in total, each intercepting a different Javascript operation applied to a proxy. However, some traps can be expressed in terms of the semantics of other traps (i.e. the semantics of the operation which they intercept is subsumed by the semantics of some other operation). We refer to such traps as *derived* traps, since their implementation may be derived from other, more *fundamental* traps.

4.7.1 Efficiency: the case for derived traps

Our handler API is comprised of 7 fundamental traps and 5 derived traps. If 7 traps are sufficient to faithfully emulate an object, why define 12 of them? The reason is efficiency: implementing a derived trap directly may be more efficient than using its default implementation based on the corresponding fundamental trap. In our particular design, as a rule of thumb, we introduced derived traps only if their implementation could intercept the operation with fewer allocations than the corresponding implementation based on the fundamental trap.

As a concrete example, consider the expression `name in obj`. The `in` operator returns a boolean indicating whether `obj` defines or inherits a property named `name`. If `obj` is a proxy, this operation triggers the handler's `has(name)` trap, which should return a boolean. `has` is a derived trap: its semantics can be directly derived from the fundamental trap `getPropertyDescriptor` which, given a property name, returns either a property descriptor for a property corresponding to that name if it exists, or returns `undefined` otherwise. Clearly, `has` could have just been defined implicitly as:

```
has: function(name) {
  return this.getPropertyDescriptor(name) !== undefined;
}
```

The downside of this implementation is that `getPropertyDescriptor` must needlessly allocate a property descriptor object if the named property exists. A custom implementation of `has` can often avoid this allocation.

While defining the intercession API, we found that there is another force at play that can prevent designers from introducing a derived trap, even if it were more efficient.

4.7.2 Consistency: the case against derived traps

One reason not to introduce a derived trap, even in the face of improved efficiency, is that introducing the distinction may break language invariants that may surprise the language runtime and programmers.

Once the API provides a proxy handler with both a fundamental and a derived trap, there is no guarantee that a proxy handler will effectively make sure that the derived trap is indeed semantically equivalent to the canonical derived implementation. For example, if a proxy handler is given the option to override both the `has` and the `getPropertyDescriptor` traps, it is not enforced that these traps return mutually consistent results. Without proxies, the programmer can count on the invariant that `name in obj` is equivalent to `Object.getPropertyDescriptor(obj, name) !== undefined`. If `obj` is a proxy, this invariant is no longer guaranteed: `has` could return true and `getPropertyDescriptor` could nevertheless return `undefined`.

For proxies, both the language runtime and programmers can no longer count on such invariants being upheld, although a "correct" handler should attempt to uphold them. Even if the handler implementor does not intend to violate the invariants, the distinction between fundamental and derived traps nevertheless introduces a maintenance cost: if either of the two traps is changed or gets overridden in a delegating object, the other trap should be changed or overridden in a compatible way.

The requirement of mutual consistency between operations is an opposing force in the decision of whether or not to introduce a derived trap for a certain operation. Whether the benefits of efficiency outweigh the mutual consistency of operations is, again, based on the particulars of the system and there is no general answer.

Fundamental versus Derived Traps: an intercession API faces a tradeoff between defining a minimal API consisting only of fundamental traps versus defining a more complex API that introduces derived traps. Derived traps may be more efficient, but may break invariants that hold between the operations corresponding to the fundamental and the derived trap.

4.8 Handler Encapsulation

A Proxy-based intercession API encapsulates its handler if, given a reference to a proxy, one cannot gain direct access to the proxy's handler. For our proxy API, this is the case. If a handler can be encapsulated behind its proxy, the handler can ensure that its traps can only ever be invoked by manipulating its corresponding proxy.

If the handler explicitly wants to be accessible from its proxy, it can cater to such access itself without violating stratification. The handler could register the association between its proxy and itself in a registry. Objects with access to this registry and to the proxy can then look up the handler, using the proxy as a key. Such a registry plays a similar role in our architecture as mirror factories in a mirror-based architecture (see Section 7.2).

4.9 Uniform Intercession

Eugster [8] introduces the term *uniform proxies* to denote an object model in which objects of all types can be proxified. More generally, we state that an intercession API is uniform if it enables the creation of proxies for all possible types of values in the language. Javascript proxies do not support uniform intercession, since they are only able to proxy for objects (object proxies) or functions (function proxies). Primitive values cannot be proxified.

The advantage of having a uniform intercession API is that it does not restrict programmers to use only language values that can be proxied. Conversely, it enables meta-programmers to potentially use the intercession API on more base-level objects.

5. Access Control Wrappers

We now show how to apply our Proxy API to implement generic wrappers for access control purposes.

5.1 Revocable Object 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 it hands out to `Bob`. In other words, `Alice` wants to have the ability to revoke `Bob`'s access to `Carol`.

One can implement this pattern of access control by wrapping `Carol` in a forwarder that can be made to stop forwarding. Such a forwarder is also known as a *caretaker* [18]. 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[2]:

```
function makeRevocableRef(target) {
  var enabled = true;
  return Object.freeze({
    caretaker: Proxy.create({
      get: function(rcvr, name) {
        if (!enabled) { throw new Error("revoked"); }
        return target[name];
      },
      has: function(name) {
        if (!enabled) { throw new Error("revoked"); }
        return name in target;
      },
      // ... and so on for all other traps
    }, Object.getPrototypeOf(target)),
    revoker: Object.freeze({
      revoke: function() { enabled = false; }
    })
  });
}
```

[2] For brevity, we left out the case in which `target` is a function, requiring the caretaker to be a function proxy.

Given this abstraction, `Alice` can now wrap `Carol` in a revokable reference, pass the caretaker to `Bob`, and hold on to the revoker such that access can later be revoked.

The caretaker is a proxy whose handler is very similar to the generic forwarder defined in Section 4 except that it explicitly checks the `enabled` flag before forwarding. As noted in Section 4.5, 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 makeRevocableRef(target) {
  var enabled = true;
  var fwdHandler = makeForwardingHandler(target);
  var baseHandler = Proxy.create({
    get: function(rcvr, name) {
      if (!enabled) { throw new Error("revoked"); }
      return fwdHandler[name];
    }
  });
  return Object.freeze({
    caretaker: Proxy.create(baseHandler,Object.getPrototypeOf(target)),
    revoker: Object.freeze({
      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`, an unwrapped reference may be exposed to `Bob`, circumventing the caretaker. This is an instance of the *two-body problem* [8], 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.

5.2 Membranes

A membrane is an extension of a caretaker that transitively imposes revocability on all references exchanged via the membrane [13]. In the following implementation, we make use of meta-level funneling once more to centralize the access control in a single trap:

```
function makeMembrane(initTarget) {
  var enabled = true;
  function wrapFunction(f) {
    return function() { // variable-argument function
      if (!enabled) {throw new Error("revoked");}
      try {
        return wrap(f.apply(wrap(this), toArray(arguments).map(wrap)));
      } catch (e) { throw wrap(e); }
    }
  }
  function wrap(target) {
    // primitives provide irrevocable knowledge, no need to wrap them
    if (isPrimitive(target)) { return target; }
```

```
    var fwdHandler = makeForwardingHandler(target);
    var baseHandler = Proxy.create({
      get: function(rcvr, name) {
        return wrapFunction(fwdHandler[name]);
      }
    });

    if (typeof target === "function") {
      var wrappedF = wrapFunction(target);
      return Proxy.createFunction(baseHandler, wrappedF);
    } else {
      return Proxy.create(baseHandler,
                          wrap(Object.getPrototypeOf(target)));
    }
  }
  return Object.freeze({
    wrapper: wrap(initTarget),
    revoker: Object.freeze({
      revoke: function() { enabled = false; }
    })
  });
}
```

A membrane consists of one or more wrappers. Every such wrapper is created by a call to the `wrap` function. Note that 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.

Recall that this example makes use of meta-level funneling. The `baseHandler` is itself a proxy with a handler. This handler need only implement a `get` trap that returns a meta-level trap (a function) that will immediately be invoked. To be able to wrap the arguments of these invocations, the `get` trap returns a wrapped version of the default forwarding trap contained in the `fwdHandler`.

If the object to be wrapped is a function (or a method, because Javascript methods are just functions), a function proxy is returned whose call trap is a wrapped version of the function, as defined by `wrapFunction`. The variable-argument function[3] returned by `wrapFunction` performs the actual wrapping of values that cross the membrane: arguments to methods cross the membrane in one direction, returned values or thrown exceptions in the other direction.

The membrane abstraction in Javascript enables innovative new ways of composing 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 [16], loading the untrusted code using a membrane-wrapped `eval` function makes it possible to isolate scripts from one another and from their container page.

Identity-preserving Membranes One problem with the above membrane abstraction is that it fails to uphold object identity across both sides of the membrane. If an object `o` is passed through the same membrane twice, an object on the other side of the membrane will receive two distinct wrappers `w1` and `w2`, such that `w1 !== w2`, even though both wrap an identical object. Likewise, passing `w1` or `w2` back through the membrane does not unwrap them but rather wraps them again. These limitations can be addressed by having the membrane maintain two maps. The first maps objects to their wrappers, such that only one canonical wrapper is created per object. The second maps wrappers back to the objects they wrap, allowing these objects to be unwrapped when they cross the membrane in the other direction.

Unfortunately, implementing these maps in Javascript is impossible without introducing memory leaks. Weak maps are a proposed language feature for the upcoming ECMAScript standard that address this issue [14]. A weak map is like a non-enumerable weak-key identity hashtable, but it avoids a crucial memory leak when cyclic dependencies exist between the keys and the values stored in the table.

[3] In Javascript, `arguments` is an array-like object that contains all arguments passed to a function. It is used to define variable-argument functions, and it is customary to convert this object into a proper array first.

	Proxies Disabled	Proxies Enabled			
		(non-proxy)		(proxy)	
Operation	a	b	$c = b/a$	d	$e = d/b$
typeof	.38 μs	.39	**1.02**	.39	**1.00**
instanceof	.49 μs	.51	**1.04**	.52	**1.01**
call	.47 μs	.47	**1.00**	.61	**1.29**
delete	.42 μs	.42	**1.00**	.71	**1.69**
has	.41 μs	.41	**1.00**	.73	**1.78**
set	.40 μs	.41	**1.02**	.74	**1.80**
get	.38 μs	.39	**1.03**	.73	**1.87**
enumerate	1.58μs	1.58	**1.00**	4.0	**2.53**

Table 2. Micro-benchmarks measuring overhead of the Proxy API (c) and of a proxy wrapper (e).

6. Implementation

Andreas Gal has implemented a prototype of our proposed Proxy API as an extension of Tracemonkey. Tracemonkey is an adaptation of Spidermonkey, Mozilla's Javascript engine, that employs trace trees for aggressive optimization [10][4].

We used the prototype implementation to get an initial idea of the overhead of our Proxy API. To this end, we performed the following micro-benchmark: first, we measured the runtime of operations applied to regular objects in regular Tracemonkey, without the Proxy API extensions. Next, we measured the runtime of the same operations on regular, *non-proxy* objects in the Tracemonkey extension with support for proxies. Finally, we measured the runtime of the operations on a *proxy* object whose handler is the default forwarding handler from Section 4.

Table 2 shows results for a subset of the operations, obtained on a MacPro Dual-core Intel Xeon (2.66Ghz) with 4GB Memory, running Mac OS X 10.5.8, using the following version of the Tracemonkey prototype: `http://hg.mozilla.org/tracemonkey/file/eba4f78cdca4`. Runtimes shown are an average over 100 runs. We should note

[4] At the time of writing, proxies are available in Tracemonkey and in the Firefox 4.0 betas at `http://www.mozilla.com/en/firefox/beta/`

that these results are obtained from a non-optimized prototype implementation. The final tracemonkey implementation may generate significantly different results.

The results in column (c) show that the addition of proxies in the virtual machine introduces no significant overhead for non-proxy objects. Hence, proxies only introduce a performance penalty when they are needed.

The results in column (e) show the overhead of using proxies to implement generic wrappers. Traps are ordered from least to most overhead. Our results confirm that operations like `typeof` and `instanceof` that do not consult the handler introduce no overhead. The overhead of intercepting the other operations, save enumeration, is less than a factor of 2. The overhead comes from the additional invocation of the handler method. Presumably, the `get` and `set` traps introduce the most overhead because they are the most optimized for non-proxy objects. Enumeration is the only operation that is more than twice as slow when applied to proxies. This is because enumeration must now proceed by iterating over a user-created array returned by the `enumerate` trap, rather than using the built-in iteration algorithm.

7. Related Work

7.1 OO Intercession APIs: a comparison

In this section we summarize a variety of message-based, object-oriented intercession APIs. We discuss whether and how each of them upholds the design principles put forth in Section 4. We do not claim that our survey is exhaustive, but we believe the most representative intercession APIs are covered. We briefly discuss the surveyed mechanisms and then summarize which of our design principles they uphold in Table 3.

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(
  aClassloader, new Class[] { Foo.class }, h);
```

Here, `proxy` implements the `Foo` interface. `h` is an object that implements a single `invoke` method. All method invocations on `proxy` are reified by calling the `h.invoke` method.

The main difference between Java proxies and Javascript proxies is that Java proxies can only intercept 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.

The Java Proxy API is non-uniform: proxies can only be constructed 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 [8] 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* [17]. AmbientTalk is a distributed dynamic language, with a mirror-based meta-level 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.

E is a secure, distributed dynamic language [12]. In E, everything is an object, but there are two kinds of object references: *near* and *eventual* references. 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 [15]:

```
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 an actual, so-called *resolved* reference. Once a reference is resolved, the proxy is bypassed and the handler no longer consulted. It fulfills a role similar to the `fix` trap described in Section 4.4. The `state` argument to `makeProxy` determines the state of the reference. The details are outside the scope of this paper, but note that this allows the eventual reference proxy to determine its state without consulting the handler, similar to how Javascript proxies can determine their prototype without consulting the handler.

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

Ducasse [6] 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* [4]. 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 Table 3 shows that each of the aforementioned intercession APIs has its own set of characteristics. It is difficult to declare one API to be better than another one. By their very nature, meta-level APIs are strongly tied to their particular language. The tradeoffs made by these APIs cannot be understood without reference to the idiosyncratic constraints and motivations of their respective languages. However, for an OO intercession API to be called robust, we believe it should adhere to the principle of stratification.

7.2 Mirrors

This work is heavily influenced by the work on mirrors. Bracha and Ungar [2] discuss the design principles of mirror-based meta-level architectures. 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 and self-modification.

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). As a point in case, consider the difficulty of defining a local mirror for a remote object. In Java, the `getClass()` method, when applied to a remote object proxy would invariably break the abstraction, as `getClass()` would reveal the proxy's class.

Mirrors are meta-level objects that are manufactured by objects known as mirror factories. To acquire a mirror on an object `o`, one does not ask `o` for its mirror but rather asks a mirror factory for a mirror on `o`. The logic of what kind of mirror to associate with an object is not tied to that object's representation. As discussed in Section 4.8, our Proxy API does not enable clients of proxies to directly access the handler through the proxy. If such access is required, the cleanest solution is to introduce a separate registry abstraction which can be regarded as a "handler factory".

Systems with a mirror-based architecture include Self [20], Strongtalk [1], Newspeak [3] and the Java Debugger API. Each of these architectures supports introspection and self-modification, but they have very limited support for intercession. To the best of our knowledge, AmbientTalk's meta-level architecture based on *mirages* [17] was the first to reconcile mirrors with support for intercession. Mirages directly inspired our work on Javascript proxies. A mirage is very much like our proxy. Its behavior is controlled by a separate handler, which in AmbientTalk is also a mirror. The main difference between AmbientTalk and Javascript in this regard, is that Javascript has no built-in notion of mirrors or mirror factories.

API	Values that can be virtualized	Stratification	Temporary Intercession	Meta-level shifting	Meta-level funnelling	Selective interception	Fundamental vs. Derived Traps	Handler Encapsulation	Uniform intercession
Javascript proxies	Objects and functions, not primitives	Yes, proxy versus handler	Yes, by fixing the proxy	Yes, if handler is itself a proxy	Yes, via the `get` trap, for a meta-handler	Yes, identity, prototype and `typeof` are fixed	Yes, see Table 1	Yes	No, primitives can't be virtualized
Java proxies	Objects whose static type is an interface or `Object`, not primitives	Yes, `Proxy` versus `Invocation-Handler`	No, proxies remain proxies forever	Yes, if handler is itself a proxy	Yes, via the `invoke` trap, for any handler	Yes, identity, class, interfaces and final methods inherited from `Object`	No, `invoke` is the only trap	No, because of `Proxy.get-Invocation-Handler(proxy)`	No, instances of non-interface classes and primitives can't be virtualized
Eugster's Uniform Java proxies	Objects, not primitives	Yes, `Proxy` versus `Invocation-Handler` and `AccessHandler`	No, proxies remain proxies forever	Yes, if invocation- and accesshandler are themselves a proxy	Yes, via the `invoke` trap, for a meta-invocation- and accesshandler	Yes, identity, class and interfaces are fixed (final fields/methods can be virtualized)	No, all traps are fundamental	No, because of `Proxy.get-Invocation-Handler(proxy)`	No, primitives can't be virtualized
AmbientTalk mirages	Objects and eventual references	Yes, mirage versus mirror	No, mirages remain mirages forever	Yes, if mirror is itself a mirage	Yes, via the `invoke` trap, for a meta-handler	Yes, an object's mirror is determined by a mirror factory, not by the mirage	No, all traps are fundamental	Yes, but can access the mirror indirectly via a mirror factory	Yes, all values are objects or references (mirages virtualize both)
E proxies	Eventual references	Yes, proxy versus `ProxyHandler`	Yes, by *resolving* the proxied reference into a real reference	No, handler cannot itself be an eventual reference proxy	Yes, if handler is a non-methodical object, via its `match` clause	Yes, identity and the "state" of a reference are fixed	Yes, `sendOnly` can be derived from `send`	Yes	Yes, all values are objects or references (proxies virtualize references)
E non-methodicals	Objects	Yes, non-methodicals have no behavior other than their `match` clause	No, non-methodicals remain non-methodicals forever	No, non-methodicals have no handler	Yes, via the `match` clause	Yes, identity and "trademarks" (unforgeable types) are fixed	No, the `match` clause is the only trap	Yes	Yes, all values are objects or references (non-methodicals virtualize objects)
Smalltalk `doesNot-Understand:`	Objects	No, proxy and handler are the same object	Yes, by replacing the object using Smalltalk's `become:` primitive	No, cannot trap invocations of `doesNot-Understand:` itself	Yes, via `doesNot-Understand:` itself, if class implements or inherits no other methods	Yes, calls to methods implemented in or inherited by the defining class will not be intercepted	No, the `doesNot-Understand:` method is the only trap	No, proxy and handler are the same object	Yes, all values are objects
ParcPlace Smalltalk dispatching classes	Objects	Yes, dispatching classes define only `dispatch-Message:`	Yes, by replacing the object using Smalltalk's `become:` primitive	Yes, by sending a `dispatch-Message:` explicitly to a dispatching class	Yes, via the `dispatch-Message:` method	No, since all interaction is via messages, and all messages are trapped	No, the `dispatch-Message:` method is the only trap	Yes	Yes, all values are objects

Table 3. Comparison of object-oriented intercession APIs

7.3 Partial Behavioral Reflection

Partial Behavioral Reflection (PBH) [19] 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, Javascript proxies only reify while they are in a trapping state. Once fixed, the proxy no longer reifies operations. 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.

7.4 CLOS

The Common Lisp Object System (CLOS) has an extensive MOP [11]. 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.

8. Conclusion

This paper introduces an intercession API for Javascript based on proxies, enabling the creation of generic wrappers and virtual objects. We characterize our API as *robust*, primarily due to its stratified design and because it upholds important invariants (pertaining to e.g. identity, runtime types and tamper-proofness).

The second contribution of this paper is the identification of general design principles for proxy-based intercession APIs. These design principles can be used to characterize similar APIs of other programming languages, as summarized in Table 3. The design principles put forward are:

Stratification: traps are defined on a separate handler.
Temporary intercession: proxies can become fixed.
Meta-level shifting: handlers as proxies shift meta-levels.
Meta-level funnelling: the ability to funnel all operations through a single trap.
Selective interception: some operations are not trapped or their semantics is determined at proxy-creation time.
Derived traps can be more efficient than **fundamental traps** but introduce the potential for inconsistencies.
Handler encapsulation: a proxy encapsulates its handler.
Uniform intercession: every value can be proxified.

Acknowledgments

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 to Brendan Eich, Andreas Gal and Dave Herman for detailed comments and to Andreas Gal for his work on the Tracemonkey implementation. Thanks to the anonymous referees for comments on an earlier draft of this article.

References

[1] G. Bracha and D. Griswold. Strongtalk: typechecking smalltalk in a production environment. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, NY, USA, 1993. ACM. ISBN 0-89791-587-9. doi: http://doi.acm.org/10.1145/165854.165893.

[2] G. Bracha and D. Ungar. 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*, pages 331–343, 2004.

[3] G. Bracha, P. Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in newspeak. In *ECOOP '10: Proceedings of the 24th European Conference on Object Oriented Programming, Maribor, Slovenia*, LNCS. Springer Verlag, June 2010.

[4] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 396–417, London, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6.

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

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

[7] ECMA International. *ECMA-262: ECMAScript Language Specification*. ECMA, Geneva, Switzerland, fifth edition, De-

cember 2009. URL http://www.ecma-international.org/publications/standards/Ecma-327.htm.

[8] P. Eugster. Uniform proxies for java. In *OOPSLA '06: Proceedings of the 21st annual conference on Object-oriented programming systems, languages, and applications*, pages 139–152, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: http://doi.acm.org/10.1145/1167473.1167485.

[9] B. Foote and R. E. Johnson. Reflective facilities in smalltalk-80. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 327–335, NY, USA, 1989. ACM. ISBN 0-89791-333-7. doi: http://doi.acm.org/10.1145/74877.74911.

[10] A. Gal. *Efficient bytecode verification and compilation in a virtual machine*. PhD thesis, University of California, Irvine, Long Beach, CA, USA, 2006.

[11] G. Kiczales, J. D. Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991. ISBN 0262111586.

[12] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, pages 195–229. Springer, 2005.

[13] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, John Hopkins University, Baltimore, Maryland, USA, May 2006.

[14] M. S. Miller. Weak maps proposal for ECMAScript Harmony, 2010. tinyurl.com/weak-maps.

[15] M. S. Miller and K. Reid. Proxies for eventual references in E, 2009. wiki.erights.org/wiki/Proxy.

[16] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript, June 2008. tinyurl.com/caja-spec.

[17] S. Mostinckx, T. Van Cutsem, S. Timbermont, and E. Tanter. Mirages: Behavioral intercession in a mirror-based architecture. In *Proceedings of the Dynamic Languages Symposium - OOPSLA'07 Companion*, pages 222–248. ACM Press, 2007.

[18] D. D. Redell. *Naming and Protection in Extensible Operating Systems*. PhD thesis, Department of Computer Science, University of California at Berkeley, Nov 1974.

[19] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. 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*, pages 27–46. ACM, 2003.

[20] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 227–242. ACM Press, 1987. ISBN 0-89791-247-0. doi: http://doi.acm.org/10.1145/38765.38828.