On the Proxy Identity Crisis

Matthias Keil   Peter Thiemann

University of Freiburg, Germany
{keilr,thiemann}@informatik.uni-freiburg.de

1. Introduction

A proxy, or wrapper, is an object that mediates access to an arbitrary target object. Proxies are widely used to perform resource management, access remote objects, impose access control [1, 5], restrict the functionality of an object [6], or to enhance the interface of an object. Ideally, a proxy is not distinguishable from other objects so that running a program with an interposed proxy should lead to the same outcome as running the program with the target object, unless the proxy imposes restrictions.

Proxies introduce a subtle problem. Because a target object may have any number of proxy objects, which are all different from the target, a single target object may obtain multiple identities—it suffers from schizophrenia! Even worse, it turns out that there is no single cure for this schizophrenia because the desired behavior depends on the use case.

Unfortunately, current proxy implementations are committed to particular use cases, which makes it hard to adapt them to uses with different requirements. We discuss two such use cases in the context of the JavaScript proxy API [1], identify its shortcomings, and propose a solution.

1.1   JavaScript Proxies

The JavaScript proxy API [1] provides a proxy constructor that takes the proxy's target object and a handler object:

```
1 var p = new Proxy (target, handler);
```

The handler object provides optional trap methods that are invoked when operations are applied to the proxy. For example, a property get like `p.foo` invokes the trap `handler.get(target,'foo',p)` if that trap is present. Untrapped operations are forwarded to the `target` object.

The JavaScript proxy API treats proxies as *opaque*: each proxy object has its own identity different from all other (proxy) objects and this difference is observable with the JavaScript equality operators `==` and `===`. When applied to two objects, both operators compare the object references.[1] The use of equality has one consequence: comparing distinct proxies returns false even though the underlying target is the same. Similarly, an unwrapped target object is not equal to any of its proxies.

1.2   Use Case: Access Control

JavaScript proxies implement access control wrappers like revocable references and membranes in a library [1]. The idea of a revocable reference is to only ever pass a proxy to an untrusted piece of code, e.g., a mashup. Once the host application deems that the mashup has finished its job, it revokes the reference which detaches the proxy from its target. Membranes extend this method recursively to all objects reachable from the object passed to a mashup. Opaque proxies are required for implementing this library.

The JavaScript proxy API is tailored to uses where access is strictly compartmentalized. The host application only sees the original objects whereas the mashup only sees proxies. Furthermore, the implementation of revocable references and membranes ensures that there is at most one proxy for each original object. For this reason, each compartment has a consistent view where object references are unique.

1.3   Use Case: Contracts

Proxies implement contracts in Racket [6] and in JavaScript [3, 5]. Contracts impose restrictions that the programmer regards as preconditions for the correct execution of a program. For example, a contract may require a method to be called with a particular type or an object property to always contain positive numbers.

During maintenance, the programmer may add contracts to a program as understanding improves. Clearly, the addition of a new contract must not change a program execution that respects the contract already. In this scenario, the program executes in a mix of original objects and proxy objects. Furthermore, there may be more than one proxy (implementing different contracts) for the same target. If introducing proxies affected the object identity, then some true comparisons would flip to false, thus changing the semantics.

Consequently, the Racket implementation provides *transparent* proxies [6], which are indistinguishable from their target object, recursively.

1.4   Assessment

Neither the opaque nor the transparent proxy implementation can be labeled as right or wrong without further qualification. Each is appropriate for a particular use case and leads to undesirable behavior in another use case.

---

[1] If one argument has a primitive type, `==` attempts to convert the other argument to the same primitive type, whereas `===` returns false if the types are different. If both arguments are objects, then both operators do the same.

It is also clear that the behavior of equality is not something the should be left to the whim of the programmer. For example, equality on objects should be an equivalence relation, which means that the equality operations == and === must not be trapped [2].

Thus, the current state of affairs in JavaScript is fully justified, but it is not well suited to implement contract systems. Hence, we explore some alternative designs that would suit both use cases.

2. Alternative Designs

Proxy-aware equality   One way to obtain transparent proxies with an implementation of opaque proxies is to provide proxy-aware equality functions like `Proxy.isEqual()` and `Proxy.isIdentical()` to replace all uses of == and ===, respectively, in an application program. This approach preserves the previous behavior and retains the possibility to distinguish proxies from target objects in library code implementing proxy abstractions. However, it would require the application code to be transformed (at run time to support `eval`), which is not feasible in an application like access control [5] that must work with unmodified foreign code.

Transparent Proxies   Making proxies generally transparent makes it impossible to test whether a reference is a proxy or an original object. However, there are abstractions that require such a test. For example, our implementation of access permissions [5] extracts the current permission from a proxy to construct a new proxy with an updated permission. This improves the efficiency of the implementation, which would otherwise generate long chains of proxy objects.

Thus, for implementing proxy abstractions it must be possible to break the transparency.

More equality operators   Another possible solution would be to reinterpret the JavaScript equality operators == and === as proxy-transparent and introduce new variants, say, `:==:` and `:===:` for their opaque cousins. The former operators are supposed to be used in application code whereas the implementation of proxy abstractions could make use of the opaque operators where needed.

No code transformation is required with this approach. However, it is not clear how to ensure that application code does not use the opaque operators. It is not even clear if it *should not* use them. While proxy abstractions can be implemented, the distinction between application and library seems too rigid. Given both operations, application code can test if one object is a proxy for another:

```
1  var isProxy = ((objA==objB) != (objA:==:objB));
```

Trapping the equality operation   We already discussed that trapping the equality operation is not appropriate. However, there is a twist that enables modifying the equality without destroying its properties. Essentially, the handler is extended with a boolean trap:

```
isTransparent : function () -> boolean
```

If the handler's trap returns false or if it is not present, the associated proxy behaves opaquely, otherwise it behaves transparently. The implementation is an extension of the equality comparison in the VM. Before testing reference identity as the last step in a comparison of two objects, the equality comparison calls a new internal `GetEqualityObject` method. For a standard object, this method returns its receiver. For a proxy object, if is `isTransparent()` on the handler returns false, then `GetEqualityObject` returns the reference to the current object. Otherwise, it recursively invokes `GetEqualityObject` on the proxy's target. For consistency, the `GetEqualityObject` method also needs to be called in other computations that depend on object identity, for instance the WeakMap abstraction provided by some JavaScript implementations.

This design enables both scenarios described in Sections 1.2 and 1.3 by configuring the handler appropriately. It also guarantees that equality is an equivalence relation in application code that does not have access to the handlers. To implement proxy-based abstractions, it may be necessary to temporarily make proxies opaque. But opaqueness can be obtained by reconfiguring the handler in the library code, analogous to the implementation of revocable references. To maintain consistency at the application level, it may be necessary to restrict modifications to this configuration to a certain scope analogously to dynamic variables [4].

3. Conclusion

We have shown that neither the transparent nor the opaque implementation of proxies is appropriate for all use cases. We discuss several amendments and propose a flexible solution that enables applications requiring transparence as well as opacity. We are currently implementing this solution in a JavaScript VM and expect to report results soon.

References

[1] T. V. Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *DLS*, pages 59–72. ACM, 2010.

[2] T. V. Cutsem and M. S. Miller. Trustworthy proxies - virtualizing objects with invariants. In *ECOOP*, volume 7920 of *LNCS*, pages 154–178. Springer, 2013.

[3] T. Disney. Contracts for JavaScript and CoffeeScript, 2011.

[4] D. R. Hanson and T. A. Proebsting. Dynamic variables. In *PLDI*, pages 264–273. ACM, 2001. ISBN 1-58113-414-2.

[5] M. Keil and P. Thiemann. Efficient dynamic access analysis using JavaScript proxies. In *DLS'13*, pages 49–60. ACM, 2013.

[6] T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. In *OOPSLA*, pages 943–962. ACM, 2012.