Trustworthy Proxies
Virtualizing Objects with Invariants*

Tom Van Cutsem[1]** and Mark S. Miller[2]

[1] Vrije Universiteit Brussel, Belgium
[2] Google Research, USA

Abstract. Proxies are a common technique to virtualize objects in object-oriented languages. A proxy is a placeholder object that emulates or wraps another target object. Both the proxy's representation and behavior may differ substantially from that of its target object.
In many OO languages, objects may have language-enforced invariants associated with them. For instance, an object may declare immutable fields, which are guaranteed to point to the same value throughout the execution of the program. Clients of an object can blindly rely on these invariants, as they are enforced by the language.
In a language with both proxies and objects with invariants, these features interact. Can a proxy emulate or replace a target object purporting to uphold such invariants? If yes, does the client of the proxy need to trust the proxy to uphold these invariants, or are they still enforced by the language? This paper sheds light on these questions in the context of a Javascript-like language, and describes the design of a Proxy API that allows proxies to emulate objects with invariants, yet have these invariants continue to be language-enforced. This design forms the basis of proxies in ECMAScript 6.

Keywords: Proxies, Javascript, reflection, language invariants, membranes

1 Introduction

Proxies are a versatile and common abstraction in object-oriented languages and have a wide array of use cases [1]. Proxies effectively "virtualize" the interface of an object (usually by intercepting all messages sent to the object). One common use case is for a proxy to *wrap* another target object. The proxy mostly behaves identical to its target, but augments the target's behavior. Access control wrappers, profilers, taint tracking [2] and higher-order contracts [3] are examples. Another common use case is for a proxy to *represent* an object that is not (yet) available in the same address space. Examples include lazy initialization of objects, remote object references, futures [4] and mock-up objects in unit tests.

Objects in object-oriented languages often have *invariants* associated with them, i.e. properties that are guaranteed to hold for the entire lifetime of the object. Some of these invariants may be enforced by the programming language itself. We will call

* Draft manuscript. To appear in Proceedings of ECOOP'13.
** Postdoctoral Fellow of the Research Foundation, Flanders (FWO).

these *language invariants*. Examples of such language invariants include immutable object fields, which are guaranteed to point to the same value throughout the object's lifetime; or, in a prototype-based language, an immutable "prototype" link pointing to an object from which to inherit other properties. Clients of an object can blindly rely on these invariants, as they are enforced by the language (that is: the language provides no mechanism by which the invariants can ever be broken). Such invariants are important for developers to reason about code, critical for security, and useful for compilers and virtual machines.

In a language with both proxies and objects with language invariants, these features interact. Can a proxy virtualize an object with language invariants? Can it wrap a target object with language invariants and claim to uphold these invariants itself? If so, does the client of the proxy need to trust the proxy to uphold these invariants, or are they still enforced by the language? This paper sheds light on these questions in the context of Javascript, a language that features both proxies and objects with language invariants.

Contribution We study the apparent tradeoff between the needs of a powerful interposition mechanism that may break language invariants versus the desire to maintain these invariants. The key contribution of this paper is the design of a Proxy API that allows proxies to virtualize objects with invariants, without giving up on the integrity of these invariants (i.e. they continue to be enforced by the language). To emphasize this, we call these proxies *trustworthy*. We call out the general mechanism of our API, named *invariant enforcement*, that is responsible for this trustworthiness.

Paper outline In Section 2, we illustrate how the issues highlighted above came up in the design of a Proxy API for Javascript. We go on to describe the general principle by which our Proxy API enforces invariants. To study this mechanism in detail, we introduce λ_{TP}, an extension of the lambda calculus featuring proxies, in Section 3. We then employ proxies to build various access control abstractions in Section 4. This will allow us to discuss the advantages and drawbacks of our API (Section 5). We end with an overview of the implementation status (Section 6) and related work (Section 7).

2 Trustworthy Proxies: the case of Javascript

The questions addressed in this paper initially arose while the authors were designing a Proxy API for Javascript[3]. We first highlight the problems we encountered in combining proxies with Javascript's language invariants, and subsequently describe how the Javascript Proxy API deals with these issues.

2.1 Language Invariants in Javascript

Javascript is known as a dynamic language, primarily because its core abstraction – objects with prototype-based inheritance – is extremely flexible. Consider the following object definition:

```
var point = { x: 0, y: 0 };
```

[3] To be precise: for ECMAScript 6, the next standard version of Javascript.

This defines a `point` object with `x` and `y` properties. Javascript objects have very weak language invariants: new properties can be added to existing objects at any time (`point.z = 0;`), existing properties can be updated by external clients (`point.x = 1;`), or even deleted (**`delete`** `point.x;`). This makes it challenging for developers to reason about the structure of objects: it is always possible that third-party client objects will modify the object at run-time.

ECMAScript 5 (ES5) [5] added new primitives that allow one to strengthen the invariants of objects, making it possible to more easily reason about the structure of objects, by protecting them from unintended modifications. Figure 1 depicts a simplified state diagram of an ES5 object.

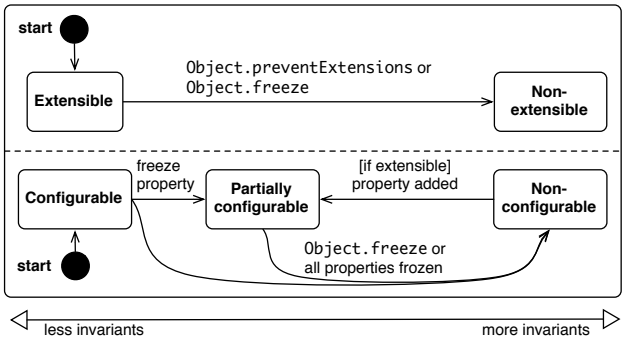

Fig. 1. Simplified state diagram of an ES5 object.

ECMAScript 5 defines two independent language invariants on objects:

Non-extensibility Every ES5 object is born in an *extensible* state. In this state, new properties can be freely added to the object. One can make an object `obj` non-extensible by calling `Object.preventExtensions(obj)`. Once an object is made non-extensible, it is no longer possible to further add new properties to the object. Once an object is non-extensible, it remains forever non-extensible.

Non-configurability Every ES5 object is born in a *configurable* state. In this state, all of its properties can be updated and deleted. It is possible to *freeze* individual properties of an object, for instance:

```
Object.defineProperty(point,"x",{writable:false,configurable:false});
```

This freezes the `point.x` property so that any attempt to update or delete that property will fail. Once a property is frozen, it remains forever frozen.

An object can evolve to become both non-extensible as well as non-configurable. Such objects are called *frozen* objects. ES5 even introduces an `Object.freeze` primitive that immediately puts an object into the frozen state, by marking the object as non-extensible, and marking all of its properties as frozen. For example, after calling

`Object.freeze(point)`, the `point` object can no longer be extended with new properties, and its existing properties can no longer be deleted or updated.

To detect whether an object is frozen, one can call `Object.isFrozen(point)`. Once the programmer has established that the `point` object is frozen, she can be sure that `point.x` will consistently refer to the same value throughout the object's lifetime.

Abstracting from the details of Javascript, these language invariants have two important characteristics: a) they are *universal*, i.e. they hold for all objects, regardless of their "type" and b) they are *monotonic*: once established on an object, they continue to hold for the entire lifetime of the object.

It are these characteristics that lend the language invariants their power to *locally* reason about objects in a dynamic language such as Javascript. Because the invariants are universal, they hold independent of the type of objects. Because the invariants are monotonic, they hold independent of pointer aliasing [6]. This monotonicity can be observed in the state diagram: there are no operations that take an object from a higher-integrity state to a lower-integrity state[4]. Hence, previously established invariants will continue to hold even if unknown parts of the program retain a reference to the object.

To focus on the essence, throughout the rest of this paper, we will simplify the exposition by combining the two ES5 language invariants (non-extensibility and non-configurability) and only considering non-frozen versus frozen objects. While we will focus on the specific invariants of frozen Javascript objects, the underlying principles should apply to any language-enforced invariant that is universal and monotonic.

2.2 The Problem: proxies can't safely uphold language invariants

Let us now consider how frozen objects interact with proxies. To this end, we will make use of a proposed Proxy API for Javascript on which we reported earlier [7]. Below is the definition of a proxy that simply forwards all intercepted "operations" (such as property access, property assignment, and so on) to a wrapped object:

```
function wrap(target) {
  var handler = {
    get: function(proxy, name) { return target[name]; },
    set: function(proxy, name, value) { target[name] = value; },
    ... // more operations can be intercepted (omitted for brevity)
  };
  var proxy = Proxy.create(handler);
  return proxy;
}
```

The function `Proxy.create` takes as its argument a `handler` object that implements a series of *traps*. Traps are functions that will be invoked by the proxy mechanism when a particular operation is intercepted on the proxy. For instance, the expression `proxy.foo` will be interpreted as `handler.get(proxy, "foo")`. Similarly, the expression `proxy.foo = 42` leads to a call to `handler.set(proxy, "foo", 42)`.

[4] A non-configurable object can be made partially configurable again by adding a new property, but only if the object is extensible. All of its existing frozen properties remain frozen.

From inspecting this code, it is clear that a proxy returned by the `wrap` function will behave the same as its `target` object. The key question is whether `Object.isFrozen(target)` then implies `Object.isFrozen(proxy)`? That is, does the proxy automatically acquire the language invariants of its target object?

The answer unfortunately is no. Whether or not the proxy still upholds the target's invariants depends on the *implementation* of the trap functions, and answering this question is in general undecidable. What is more: the fact that `proxy` is a proxy *for* the `target` object is only implicit in the above program: the `proxy` object does not even possess a direct reference to the `target` object. It just happens to be a proxy *for* it because the trap functions have closed over the `target` variable, and decided to forward the intercepted operations to the object stored in this variable. In fact, it is perfectly reasonable to create proxy objects that don't forward anything to any target object at all (the objects represented by such proxies would be entirely virtual).

Let us illustrate this point by creating an alternative version of `wrap` that probably does not uphold its target object's invariants:

```
function wrapRandom(target) {
  var handler = {
    get: function(proxy, name) { return Math.random(); },
    ...
  };
  var proxy = Proxy.create(handler);
  return proxy;
}
```

If we now call **var** `proxy = wrapRandom(Object.freeze(point))`, passing the frozen `point` object as the `target`, then `proxy.x` may yield a different number each time it is dereferenced. That is hardly the behavior one would expect of a frozen object, so `Object.isFrozen(proxy)` must clearly return false.

It turns out that having `Object.isFrozen` return false for any proxy is the only safe option. The alternative (returning true) would break the invariant that *if* an object is frozen, *then* one can expect property access to consistently return the same value, as the above example demonstrates.

By requiring `Object.isFrozen` to return false for all proxies, we have taken away some of the virtualization power of proxies, by disallowing them to ever virtualize frozen objects. This is necessary to ensure the integrity of the invariant associated with frozen objects ("properties of a frozen object are immutable").

This is an unfortunate outcome, as there certainly exist legitimate use cases where the proxy would want to appear as frozen, and where the proxy handler does behave as a frozen object. One can easily imagine a variant of the `wrap` function that does some form of profiling or contract checking, but otherwise faithfully forwards all operations to its target. Because the proxy cannot faithfully virtualize the frozen invariant of its target, transparency is lost. Such transparency may be crucial in some applications, where objects may need to be substituted by proxies without affecting client behavior.

2.3 The Solution: Invariant Enforcement

To overcome the limitation of proxies to virtualize frozen objects, we modified the initial Proxy API as follows:

- Proxies now refer directly to the target object that they wish to wrap.
- Whenever an operation is intercepted, a trap is called on the handler, as before.
- Before the trap gets to return its result to the client, the proxy verifies whether an invariant is associated with the intercepted operation on the target object.
- If so, the proxy asserts that the result is consistent with the expected (invariant) result. A failed assertion leads to a run-time exception, thus warning the client of an untrustworthy proxy.

Because the proxy now "knows" the target object that the handler wants to virtualize, the proxy has a way of *verifying* whether a) the target has an invariant and b) the trap doesn't violate that invariant. This is the key mechanism by which proxies are enforced to uphold the target object's invariants. We call this mechanism *invariant enforcement*. It is illustrated graphically in Figure 2.

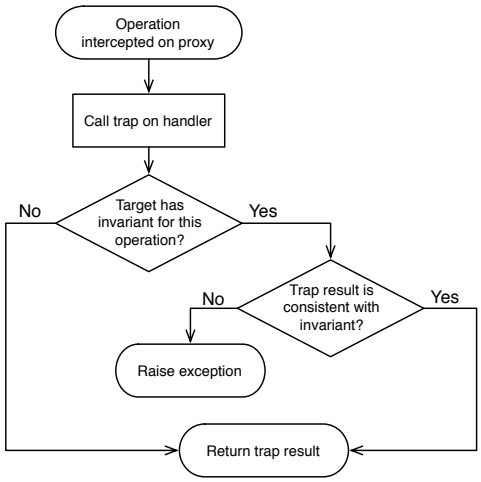

Fig. 2. Invariant enforcement in trustworthy proxies.

The `wrap` function from the previous Section can be rewritten using the new API:

```
function wrap(target) {
  var handler = {
    get: function(target, name) { return target[name]; },
    set: function(target, name, value) { target[name] = value; },
    ... // more operations can be intercepted (omitted for brevity)
  };
  var proxy = Proxy(target, handler);
```

```
  return proxy;
}
```

In this version of the API, `Proxy` has become a function of two arguments and takes as its first argument a direct reference to the `target` object for which the proxy will act as a stand-in. The handler traps remain mostly the same, except that instead of the `proxy`, the traps now receive the `target` as their first argument. Figure 3 depicts the relationship between the objects, and shows how operations like property access and update are interpreted by proxies.

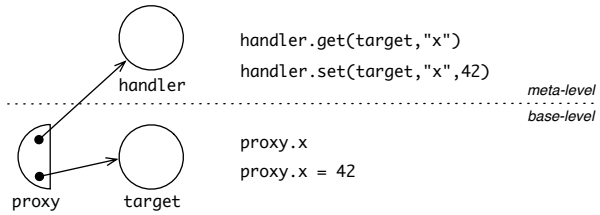

Fig. 3. Relationship between `proxy`, `target` and `handler`.

For the proxies generated by this API, it holds that `Object.isFrozen(proxy)` if and only if `Object.isFrozen(target)`. That is, such proxies can faithfully virtualize the frozen invariant of their target. And because of invariant enforcement, the outcome of the operation is actually trustworthy.

How does invariant enforcement work in the specific case of frozen objects and property access? When evaluating the expression `proxy.x`, where `proxy` refers at run-time to a proxy object, one can reason about this property access as if the proxy executed the following code:

```
// inside the proxy, intercepting proxy.x
var value = handler.get(target, "x");
if (Object.isFrozen(target)) {
  var expectedValue = target.x;
  if (expectedValue !== value) {
    throw new Error("frozen invariant violated");
  }
}
return value;
```

Note that invariant enforcement is a two-step process: first the proxy verifies whether there is an invariant to be enforced on the target (in this case: is the target frozen?). If so, then the trap result is verified. It may seem odd that the proxy must first check whether the target has invariants at interception-time. Can it not determine the target's invariants ahead of time? Unfortunately the answer is no, since, in Javascript, objects can come to acquire new invariants at run-time. For instance, an object becomes frozen only *after* a

call to `Object.freeze`. Before that time, the object is not frozen and no invariants need be enforced[5].

To summarize, by redesigning the Proxy API, the apparent tradeoff between proxies and invariants is resolved. Proxies get to virtualize objects with invariants, but only if they can provide a target object that "vouches for" these invariants, such that the result of trap invocations on the handler can be verified. Programmers, secure sandboxes and virtual machines can all continue to rely on the language invariants, regardless of whether they are dealing with built-in objects or proxies.

3 The λ_{TP} calculus

To study invariant enforcement in more detail, we now turn our attention from Javascript to a minimal calculus named λ_{TP}. While the full Javascript language is fairly large and complex, at its core lies a simple dynamic language with first-class lexical closures and concise object literal notation. This simple core is what Crockford refers to as "the good parts" [8], and it is this core that is modelled by λ_{TP}. At this stage, we should warn the reader that it has *not* been our goal to accurately formalize Javascript. For an accurate formalization of Javascript, we refer to [9].

Our goal is to faithfully model in λ_{TP} the interaction between proxy, target and handler objects as informally discussed in Section 2.3. We first introduce the core calculus with proxies that are not trustworthy. We then add support for trustworthy proxies by revising the semantics.

3.1 Core λ_{TP}

Syntax λ_{TP} is based on the untyped λ-calculus with strict evaluation rules, extended with constants, records and proxies. It is inspired by the λ_{proxy} calculus of Austin *et al.* [2], which models proxies representing virtual values (see Section 7.2).

Values of the core calculus include constants (strings s and booleans b), functions $\lambda x.e$ and records. Records are either primitive records or proxies. Primitive records are finite mutable maps from strings to values. A primitive record created using the expression $\{\overline{s : e}\}$ declares zero or more initial properties whose value can be retrieved ($e[e]$) or updated ($e[e] := e$). Proxies are created using the expression `proxy` $e\ e$ where the arguments denote the proxy's target record and handler record respectively.

Records, like Javascript objects, can be frozen. A frozen record cannot be extended with new properties and its existing properties can no longer be updated. Records can be made frozen using the `freeze` r operator, which corresponds to Javascript's `Object.freeze` primitive. The `isFrozen` r operator can be used to test whether the record r is frozen, and models Javascript's `Object.isFrozen(r)` primitive.

In Javascript one can *enumerate* the properties of an object by means of a `for-in` loop. λ_{TP} similarly features a `for` $(x : e)\ e'$ expression that iteratively evaluates the loop body e' with x bound to each property key (a string) of a record e.

[5] However, once an invariant on the target object has been established, a smart implementation could from that point on elide the check for that invariant on future intercepted operations, since the invariant will hold forever after.

The expression `keys` r eagerly retrieves all of the property keys of r. As λ_{TP} does not feature lists or arrays as primitive values, we encode the set of property keys of a record as another record mapping those keys to `true`.

The calculus further includes a `typeof` operator inspired by the corresponding operator in Javascript, revealing the type of a value as a string. The `typeof` operator classifies any value as either a function, a record or a constant.

The λ_{TP} calculus

Syntax

$e ::=$		**Expressions**
	x	variable
	c	constant
	$\lambda x.e$	abstraction
	$e\ e$	application
	`if` $e\ e\ e$	conditional
	$e = e$	equality test
	$\{\overline{s : e}\}$	record creation
	$e[e]$	record lookup
	$e[e] := e$	record update
	`for` $(x : e)\ e$	enumeration
	`proxy` $e\ e$	proxy creation
	`freeze` e	freezing
	`isFrozen` e	frozen test
	`typeof` e	type test
$c ::= s \mid$ `null` $\mid b$		**Constants**
$b ::=$ `true` $\mid$ `false`		**Booleans**

Syntactic Sugar

$e.x$	$\stackrel{\text{def}}{=}$	$e[\texttt{"x"}]$
$e.x := e'$	$\stackrel{\text{def}}{=}$	$e[\texttt{"x"}] := e'$
$x : e$	$\stackrel{\text{def}}{=}$	$\texttt{"x"} : e$
`let` $x = e\ ;\ e'$	$\stackrel{\text{def}}{=}$	$(\lambda x.e')\ e$
`var` $x = e\ ;\ e'$	$\stackrel{\text{def}}{=}$	`let` $y = \{\}\ ;\ y.x := \theta e\ ;\ \theta e'$ where $\theta = [x := y.x]$
$e\ ;\ e'$	$\stackrel{\text{def}}{=}$	$(\lambda x.e')\ e \quad x \notin FV(e')$
$e\ e'\ e''$	$\stackrel{\text{def}}{=}$	$(e\ e')\ e''$
$\lambda.e$	$\stackrel{\text{def}}{=}$	$\lambda x.e \quad x \notin FV(e)$
$\lambda x, y.e$	$\stackrel{\text{def}}{=}$	$\lambda x.\lambda y.e$
`!` e	$\stackrel{\text{def}}{=}$	`if` e `false true`
`assert` e	$\stackrel{\text{def}}{=}$	`if` e `null` (`null null`)
`keys` e	$\stackrel{\text{def}}{=}$	`let` $r = \{\}\ ;$ `for` $(x : e)\ r[x] :=$ `true` $;\ r$

Semantics The runtime syntax of λ_{TP} extends expressions with addresses a and enumerations $\mathrm{enum}\ (x : S)\ e$. The latter denote the continuation of an active record enumeration.

λ_{TP} SEMANTICS (UNTRUSTWORTHY PROXIES)

Runtime syntax

$a, t, h \in$ **Address**
$s \in S \subset$ **String**

v, w	$::=$	$c \mid a$	Values
e	$::=$	$\ldots \mid a \mid \mathrm{enum}\ (x : S)\ e$	Runtime expressions
H	$::=$	$a \rightarrow_p (\texttt{rec}\ f\ b \mid \texttt{fun}\ x\ e \mid \texttt{pxy}\ t\ h)$	Heaps
f	$::=$	$s \rightarrow_p v$	Record mapping
E	$::=$	$\bullet\ e \mid v\ \bullet \mid \texttt{if}\ \bullet\ e\ e \mid \{\overline{s : v}, s : \bullet, \overline{s : e}\}$	Evaluation contexts
	$\mid$	$\bullet = e \mid v = \bullet \mid \bullet\,[e] \mid v[\bullet] \mid \texttt{typeof}\ \bullet$	
	$\mid$	$\bullet[e] := e \mid v[\bullet] := e \mid v[w] := \bullet \mid \texttt{for}\ (x : \bullet)\ e$	
	$\mid$	$\texttt{proxy}\ \bullet\ e \mid \texttt{proxy}\ v\ \bullet \mid \texttt{freeze}\ \bullet \mid \texttt{isFrozen}\ \bullet$	

Reduction rules

$H, \{\overline{s : v}\} \rightarrow H[a \mapsto \texttt{rec}\ f\ \texttt{true}], a$	$a \notin dom(H), f(s) = v$	[ALLOCREC]
$H, \lambda x.e \rightarrow H[a \mapsto \texttt{fun}\ x\ e], a$	$a \notin dom(H)$	[ALLOCFUN]
$H, a\ v \rightarrow H, e[v/x]$	$H(a) = \texttt{fun}\ x\ e$	[APPLY]
$H, \texttt{if true}\ e_1\ e_2 \rightarrow H, e_1$		[IFTRUE]
$H, \texttt{if false}\ e_1\ e_2 \rightarrow H, e_2$		[IFFALSE]
$H, v = w \rightarrow H, \mathrm{eq}(v, w)$		[EQUAL]
$H, a[s] \rightarrow H, v$	$H(a) = \texttt{rec}\ f\ b, f(s) = v$	[GET]
$H, a[s] \rightarrow H, \texttt{null}$	$H(a) = \texttt{rec}\ f\ b, s \notin dom(f)$	[GETMISSING]
$H, a[s] := v \rightarrow H[a \mapsto \texttt{rec}\ f[s \mapsto v]\ b], v$	$H(a) = \texttt{rec}\ f\ b, b = \texttt{false}$	[SET]
$H, \texttt{for}\ (x : a)\ e \rightarrow H, \mathrm{enum}\ (x : dom(f))\ e$	$H(a) = \texttt{rec}\ f\ b$	[STARTENUM]
$H, \mathrm{enum}\ (x : \emptyset)\ e \rightarrow H, \texttt{null}$		[STOPENUM]
$H, \mathrm{enum}\ (x : S)\ e \rightarrow H, \texttt{let}\ x = s\ ;\ e\ ;\ \mathrm{enum}\ (x : S')\ e$	$s \in S,\ S' = S \setminus \{s\}$	[STEPENUM]
$H, \texttt{freeze}\ a \rightarrow H[a \mapsto \texttt{rec}\ f\ \texttt{true}], a$	$H(a) = \texttt{rec}\ f\ b$	[FREEZE]
$H, \texttt{isFrozen}\ a \rightarrow H, b$	$H(a) = \texttt{rec}\ f\ b$	[ISFROZEN]
$H, \texttt{typeof}\ a \rightarrow H, \texttt{"function"}$	$H(a) = \texttt{fun}\ x\ e$	[TYPEOFFUN]
$H, \texttt{typeof}\ a \rightarrow H, \texttt{"record"}$	$H(a) = \texttt{rec}\ f\ b$	[TYPEOFREC]
$H, \texttt{typeof}\ c \rightarrow H, \texttt{"constant"}$		[TYPEOFCST]
$H, \texttt{proxy}\ t\ h \rightarrow H[a \mapsto \texttt{pxy}\ t\ h], a$	$a \notin dom(H)$	[ALLOCPXY]
$H, \texttt{typeof}\ a \rightarrow H, \texttt{"record"}$	$H(a) = \texttt{pxy}\ t\ h$	[TYPEOFPXY]
$H, a[s] \rightarrow H, h[\texttt{"get"}]\ t\ s$	$H(a) = \texttt{pxy}\ t\ h$	[GETPXY]
$H, a[s] := v \rightarrow H, h[\texttt{"set"}]\ t\ s\ v\ ;v$	$H(a) = \texttt{pxy}\ t\ h$	[SETPXY]
$H, \texttt{freeze}\ a \rightarrow H, h[\texttt{"freeze"}]\ t\ ;a$	$H(a) = \texttt{pxy}\ t\ h$	[FREEZEPXY]
$H, \texttt{isFrozen}\ a \rightarrow H, h[\texttt{"isFrozen"}]\ t$	$H(a) = \texttt{pxy}\ t\ h$	[ISFROZENPXY]
$H, \texttt{for}\ (x : a)\ e \rightarrow H, \texttt{for}\ (x : h[\texttt{"enum"}]\ t)\ e$	$H(a) = \texttt{pxy}\ t\ h$	[ENUMPXY]
$H, E[e] \rightarrow H', E[e']$	$H, e \rightarrow H', e'$	[CONTEXT]

A heap H is a partial mapping from addresses a to three types of values: functions, primitive records or proxies. Functions are represented as $\texttt{fun}\, x\, e$ where x is the formal parameter and e is the function body. Records are either primitive records or proxies. Primitive records are represented as $\texttt{rec}\ f\ b$ where f is a partial function from strings to values representing the record's properties and b is a boolean indicating whether or not the record is frozen. Proxies are represented as $\texttt{pxy}\ t\ h$ where t is the address of the proxy's target and h is the address of the proxy's handler (both of which should denote records, not functions).

An evaluation state H, e denotes a heap H and the expression e being evaluated. The rules for the evaluation relation $H, e \rightarrow H', e'$ describe how expressions are evaluated in λ_{TP}.

The [EQUAL] rule describes equality of values v and w in terms of an `eq` primitive, which may be defined as:

$$\mathrm{eq}(v, w) \stackrel{\mathrm{def}}{=} \begin{cases} \mathrm{true} & \text{if } v \text{ and } w \text{ denote the same constant } c \\ \mathrm{true} & \text{if } v \text{ and } w \text{ denote the same address } a \\ \mathrm{false} & \text{otherwise} \end{cases}$$

This primitive represents identity-equality: constants are equal if and only if they denote the same constant value c; functions, records and proxies are equal if and only if their addresses a are equal.

The rules [GET] and [GETMISSING] together implement record lookup. If the property is not found, `null` is returned (similar to `undefined` being returned for missing properties in Javascript). The [SET] rule implements record update. If a record is updated with a non-existent property, the property is added to the record. Note that the [SET] rule only allows updates on non-frozen records. It is an error to try and add or update a property on a frozen record.

The rules [STARTENUM], [STOPENUM] and [STEPENUM] together implement property enumeration over records. There are two aspects worth noting about our enumeration semantics: first, enumeration is driven by a fixed snapshot of the record's properties. The snapshot includes those properties present when the enumeration starts. Properties added to the record while reducing a `for`-expression will not be enumerated during the same enumeration. Second, the order in which a record's properties are enumerated is left unspecified. This is also true of property enumeration in Javascript.

The rule [FREEZE] shows that the `freeze` operator, applied to a record address a always yields the same address a, but as a side-effect modifies the heap so that the record is now marked frozen, regardless of whether it was already frozen.

3.2 Untrustworthy Proxies in λ_{TP}

A proxy is created using the expression $\mathrm{proxy}\ e\ e'$ where the first expression denotes the proxy's target and the second expression denotes the proxy's handler (both expressions must reduce to an address denoting another record, i.e. either a built-in record or another proxy). λ_{TP} proxies can intercept five operations: record lookup, record update, enumeration, freezing and frozen tests. The signatures of the trap functions are shown below:

$$
\begin{aligned}
\texttt{get} &:: \mathit{target} \to \mathit{string} \to \mathit{result\ value} \\
\texttt{set} &:: \mathit{target} \to \mathit{string} \to \mathit{update\ value} \to \mathit{unit} \\
\texttt{freeze} &:: \mathit{target} \to \mathit{unit} \\
\texttt{isFrozen} &:: \mathit{target} \to \mathit{boolean} \\
\texttt{enum} &:: \mathit{target} \to \mathit{keys\ record}
\end{aligned}
$$

The rules [GETPXY] and [SETPXY] prescribe that record lookup and record update on a proxy are equivalent to calling that proxy's `get` and `set` traps. Note that the return value of the `set` trap itself is ignored: record update always reduces to the update value v. A similar observation can be made for freezing: the `freeze` operator always returns the frozen object and ignores the result of the `freeze` trap. The `isFrozen` trap, on the other hand, is expected to return a boolean indicating the result of the test.

The rule [ENUMPXY] shows that upon enumerating the properties of a proxy, that proxy's `enum` trap is called. This trap is expected to return a set of property names, encoded as a record. The returned record's properties are subsequently enumerated.

3.3 Language Invariants

While the above reduction rules capture the essence of what it means for a proxy to intercept an operation, they do not aim to uphold any language invariants. In other words: proxies, as currently specified, are not trustworthy. This is easy enough to see: there is nothing stopping a proxy from returning `true` from its `isFrozen` trap, while still reporting different values over time for its properties via its `get` trap.

In this Section, we explicitly spell out the invariants of the λ_{TP} calculus with respect to frozen *primitive* records. In the following Section, we then revise the proxy reduction rules such that proxies obey the same invariants as primitive records, thus making proxies trustworthy. The invariants of frozen primitive records are as follows:

I1 The properties of a frozen record are immutable. If `isFrozen` r = `true` then $r[s]$ always reduces to the same result value v. This also implies that if r does not have a property named s, $r[s]$ will always reduce to `null`.

I2 If the `freeze` r operator returns successfully, r is guaranteed to be frozen.

I3 Freezing is monotonic: once `isFrozen` r = `false`, it remains `false` thereafter. In other words: once frozen, a record remains forever frozen.

I4 Enumerating properties using a `for`-loop over a frozen record r always enumerates the same set of properties. That is, it enumerates at least all properties defined on r, and it does not enumerate any properties that do not exist on r.

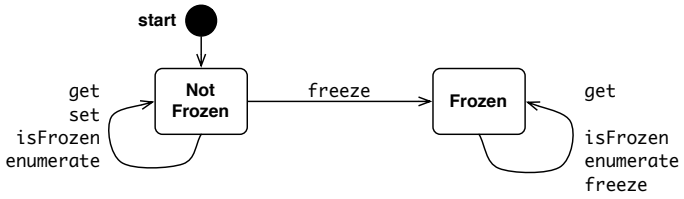

Fig. 4. State chart depicting the valid states of a λ_{TP} record.

Figure 4 depicts a state chart illustrating the state of a record, and the effect of interceptable operations on records on that state.

3.4 Trustworthy Proxies in λ_{TP}

In the semantics described thus far, proxies are *untrustworthy*, meaning that they may violate the above invariants of primitive records. Below, we introduce a set of updated reduction rules, which turn λ_{TP} proxies into *trustworthy* proxies.

λ_{TP} **SEMANTICS (TRUSTWORTHY PROXIES)**

$$
\begin{array}{rlll}
H, a[s] \rightarrow H, \texttt{let}\ x = h[\texttt{"get"}]\ t\ s\ ; & H(a) = \texttt{pxy}\ t\ h & [\textsc{GetPxy'}] \\
\quad \texttt{if}\ (\texttt{isFrozen}\ t) & & \\
\quad\quad (\texttt{assert}\ x = t[s]\ ;\ x) & & \\
\quad\quad x & & \\
H, a[s] := v \rightarrow H, h[\texttt{"set"}]\ t\ s\ v\ ; & H(a) = \texttt{pxy}\ t\ h & [\textsc{SetPxy'}] \\
\quad \texttt{assert}\ (!\ \texttt{isFrozen}\ t)\ ;\ v & & \\
H, \texttt{for}\ (x : a)\ e \rightarrow H, \texttt{let}\ y = h[\texttt{"enum"}]\ t\ ; & H(a) = \texttt{pxy}\ t\ h & [\textsc{enumPxy'}] \\
\quad \texttt{if}\ (\texttt{isFrozen}\ t) & & \\
\quad\quad \texttt{sameKeys}\ y\ (\texttt{keys}\ t) & & \\
\quad\quad \texttt{null}\ ; & & \\
\quad \texttt{for}\ (x : y)\ e & & \\
H, \texttt{freeze}\ a \rightarrow H, h[\texttt{"freeze"}]\ t\ ; & H(a) = \texttt{pxy}\ t\ h & [\textsc{freezePxy'}] \\
\quad \texttt{assert}\ (\texttt{isFrozen}\ t)\ ;\ a & & \\
H, \texttt{isFrozen}\ a \rightarrow H, \texttt{let}\ x = h[\texttt{"isFrozen"}]\ t\ ; & H(a) = \texttt{pxy}\ t\ h & [\textsc{isFrozenPxy'}] \\
\quad \texttt{assert}\ (x = \texttt{isFrozen}\ t)\ ;\ x & &
\end{array}
$$

The rule [GETPXY'] includes a post-condition that asserts whether the return value of the `get` trap corresponds to the target's value for the same property s, but only if the target is frozen. This assertions contributes to upholding invariant **I1**. Note that the proxy target t may itself be a proxy, in which case the expression $t[s]$ in the assertion will recursively trigger the [GETPXY'] rule, eventually bottoming out when the target is a primitive record.

The rule [SETPXY'] includes a post-condition that asserts that the target is not frozen. Again, this assertion ensures invariant **I1**.

The rule [FREEZEPXY'] includes a similar post-condition, this time testing whether the target is indeed frozen, if the `freeze` trap returned successfully. Clients that call `isFrozen r` expect `r` to be frozen afterwards. This guarantees invariant **I2**.

The rule [ISFROZENPXY'] inserts a post-condition that verifies whether the return value of the `isFrozen` trap corresponds to the current state of the target. Any discrepancy in the result could confuse client code: if the `isFrozen` trap is allowed to return `true` while wrapping a non-frozen target, clients would perceive the proxy as frozen while its `get` trap could still return arbitrary values, thus breaking invariant **I1**. The other way around, if the trap is allowed to return `false` while wrapping a frozen target, it may break invariant **I3** if it previously already returned `true` from its `isFrozen` trap.

The rule [ENUMPXY'] inserts a post-condition, ensuring that if the proxy wraps a frozen target, the returned set of to-be-enumerated properties corresponds to the set of properties of the target itself. This guarantees invariant **I4**. `sameKeys` is defined as follows:

$$\texttt{sameKeys}\, r_1, r_2 \stackrel{\text{def}}{=} \texttt{for}\, (x_1 : r_1)\, \texttt{assert}\, r_2[x_1] = \texttt{true}\,; \\ \texttt{for}\, (x_2 : r_2)\, \texttt{assert}\, r_1[x_2] = \texttt{true}$$

`sameKeys` checks whether two records representing sets of properties denote the same set of property keys. One can think of the first `for`-loop as checking whether all properties that the proxy enumerated are indeed properties of the frozen target, and of the second `for`-loop as checking whether the proxy did indeed enumerate all properties of the frozen target.

Note that the validity of all pre and post-condition checks depends also on the fact that the assertions compare the trap result against the expected value using the = operator, and that proxies cannot intercept this operator. Thus, proxies cannot directly influence the outcome of the assertions.

4 Access control wrappers

We now put trustworthy proxies to work by using them to build access control wrappers. Such wrappers typically perform a set of dynamic checks upon intercepting certain operations, but otherwise try to be as transparent as possible to client objects. If the check succeeds, the wrapper often simply forwards the intercepted operation to the wrapped object. Using trustworthy proxies, we can build access control wrappers that uphold the invariants of wrapped target objects, further increasing transparency for clients.

4.1 Revocable references

A revocable reference is a simple type of access control wrapper. 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` should become 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 [10]. In the absence of language-level support for proxies, the programmer is forced to write a distinct caretaker for each type of object to be wrapped. Proxies enable the programmer to abstract from the specifics of the wrapped object's interface and instead write a *generic* caretaker. Using such a generic caretaker abstraction, `alice` can hand out a revocable reference to `bob` as follows:

```
var carol = {...};
// caretaker is a tuple consisting of a proxy reference, and a revoke function
var caretaker = makeCaretaker(carol);
var carolproxy = caretaker.ref; // a proxy for carol, which alice can give to bob
```

```
bob.use(carolproxy);
// later, alice can revoke bob's access...
caretaker.revoke(); // carolproxy is now useless
```

A key point is that as long as the caretaker is not revoked, the proxy is sufficiently transparent so that `bob` can use `carolproxy` as if it were the real `carol`. There is no need for `bob` to change the way he interacts with `carol`. Indeed, if `bob` has no other, direct, reference to `carol`, `bob` is not even able to tell that `carolproxy` is only a proxy for `carol`.

Below is an implementation of the `makeCaretaker` abstraction in λ_{TP}:

$$
\begin{array}{l}
\mathrm{makeCaretaker} \stackrel{\mathrm{def}}{=} \lambda x. \\
\ \mathtt{var}\ \mathrm{revoked} = \mathtt{false}; \\
\quad \{\mathrm{ref} : \mathtt{proxy}\ x\ \{ \\
\qquad \mathrm{get} : \lambda t, s.\mathtt{assert}\ (!\mathrm{revoked})\ ; t[s] \\
\qquad \mathrm{set} : \lambda t, s, v.\mathtt{assert}\ (!\mathrm{revoked})\ ; t[s] := v \\
\qquad \mathrm{enum} : \lambda t.\mathtt{assert}\ (!\mathrm{revoked})\ ; \mathtt{keys}\ t\ \} \\
\qquad \mathrm{freeze} : \lambda t.\mathtt{assert}\ (!\mathrm{revoked})\ ; \mathtt{freeze}\ t \\
\qquad \mathrm{isFrozen} : \lambda t.\mathtt{assert}\ (!\mathrm{revoked})\ ; \mathtt{isFrozen}\ t\ \} \\
\quad \mathrm{revoke} : \lambda.\mathrm{revoked} := \mathtt{true}\}
\end{array}
$$

The argument x to `makeCaretaker` is assumed to be a record. The function returns a record `r` that pairs a proxy `r.ref` with an associated function `r.revoke`. Both share a privately scoped `revoked` boolean that signifies whether or not the proxy was previously revoked. The proxy's handler implements all traps by first verifying whether the reference is still unrevoked. If this is the case, it forwards each operation directly to the wrapped target record t.[6]

A limitation of the above caretaker abstraction is that values exchanged via the caretaker are themselves not recursively wrapped in a revocable reference. For example, if `carol` defines a method that returns an object, she exposes a direct reference to that object to `bob`, circumventing `alice`'s caretaker. The returned object may even be a reference to `carol` herself (e.g. by returning `this` from a method in Javascript). The abstraction discussed in the following section addresses this issue.

4.2 Membranes: transitively revokable references

A membrane is an extension of a caretaker that transitively imposes revocability on all references exchanged via the membrane [11].

One use case of membranes is the safe 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 [12], 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 then renders it instantly powerless.

[6] We take the notational liberty of using names like t,s and v for trap parameters, which hint at the parameters' type, rather than using strict variable names like x, y and z.

The objective of the membrane is to fully keep the object graph g created by the untrusted code isolated from the object graph g' of the containing page. When creating a membrane, one usually starts with a single object that forms the "entry point" into g. At the point when the membrane is created, it is usually assumed that apart from a single reference to the entry point of g, g and g' are otherwise fully isolated (i.e. there are no other direct references from any objects in g to any objects in g' and vice versa). If this is not the case, then the membrane will not be able to fully enclose and isolate g.

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 = { y: wetA }
var membrane = makeMembrane(wetB) // wetB acts as the entry point
var dryB = membrane.ref // a proxy for wetB
var dryA = dryB.y // references are transitively wrapped
dryA.x            // returns 1, constants are not wrapped
membrane.revoke() // revokes all dry references at once
dryB.y            // error: revoked
dryA.x            // error: revoked
```

The interface of a membrane is the same as that of a caretaker. Its implementation in λ_{TP} is shown in Figure 5. 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 `revoked` variable. Assigning the variable to `false` instantaneously revokes all of the membrane's wrappers.

$$
\begin{array}{l}
\mathrm{makeMembrane} \stackrel{\mathrm{def}}{=} \lambda x. \\
\quad \texttt{var}\ \mathrm{revoked} = \texttt{false}; \\
\quad\quad \texttt{let}\ \mathrm{wrap} = \lambda y. \\
\quad\quad\quad \texttt{if typeof}\ y = \texttt{"constant"} \\
\quad\quad\quad\quad y \\
\quad\quad\quad\quad \texttt{if typeof}\ y = \texttt{"function"} \\
\quad\quad\quad\quad\quad \lambda z.\texttt{assert}\ (!\mathrm{revoked})\ ; \mathrm{wrap}\ (y\ (\mathrm{wrap}\ z)) \\
\quad\quad\quad\quad\quad \texttt{proxy}\ y\ \{ \\
\quad\quad\quad\quad\quad\quad \mathrm{get} : \lambda t, s.\texttt{assert}\ (!\mathrm{revoked})\ ; \mathrm{wrap}\ t[s] \\
\quad\quad\quad\quad\quad\quad \mathrm{set} : \lambda t, s, v.\texttt{assert}\ (!\mathrm{revoked})\ ; t[s] := (\mathrm{wrap}\ v) \\
\quad\quad\quad\quad\quad\quad \mathrm{freeze} : \lambda t.\texttt{assert}\ (!\mathrm{revoked})\ ; \texttt{freeze}\ t \\
\quad\quad\quad\quad\quad\quad \mathrm{isFrozen} : \lambda t.\texttt{assert}\ (!\mathrm{revoked})\ ; \texttt{isFrozen}\ t \\
\quad\quad\quad\quad\quad\quad \mathrm{enum} : \lambda t.\texttt{assert}\ (!\mathrm{revoked})\ ; \texttt{keys}\ t\ \}\ ; \\
\quad\quad\quad \{\mathrm{ref} : \mathrm{wrap}\ x \\
\quad\quad\quad\ \ \mathrm{revoke} : \lambda.\mathrm{revoked} := \texttt{true}\}
\end{array}
$$

Fig. 5. Membranes in λ_{TP}.

The `wrap` function does a case-analysis on y based on the three types of values in λ_{TP}: constants are passed through a membrane unwrapped; functions are wrapped as functions that transitively wrap their argument and return value; and records are wrapped using proxies.

Although this implementation does not distinguish them, there are two "directions" in which a value can flow across the membrane: the argument z to a wrapped function and the argument v to the `set` trap are inbound, while the return value of a wrapped function and the return value of the `get` trap are outbound. In this version, both inbound and outbound values get wrapped without distinction.

Note that because the membrane faithfully forwards the `freeze` and `isFrozen` operations (as long as it is not revoked), clients on either side of the membrane can inspect whether or not the wrapped object is frozen and act accordingly. Because proxies are trustworthy, clients can have complete confidence in the outcome of the `freeze` and `isFrozen` operators, even when a membrane is interposed.

4.3 Membranes and frozen objects

The above membrane implementation works fine except for one important detail, which surfaces when a frozen record crosses the membrane. Things go wrong when code on either the dry (outside) or the wet (inside) side of the membrane tries to lookup a property $r[s]$ on a wrapper r for a frozen record t. Instead of getting back a transitively wrapped value, the program will trip on an assertion.

Recall that the invariant enforcement mechanism inserts a post-condition check upon evaluating $r[s]$ (rule [GETPXY']), testing whether its return value is equal to $t[s]$. However, the `get` trap of the above membrane abstraction always returns a fresh wrapper for the value $t[s]$ of the wrapped, frozen target t. Unless $t[s]$ is a constant, the check fails since the trap is returning a proxy for $t[s]$, not $t[s]$ itself.

To circumnavigate this issue, rather than letting the proxy wrapper directly wrap the target on the other side of the membrane, we let it wrap a *shadow target*, a dummy – initially empty – record that is initially not frozen. Figure 6 shows the initial state of such a membrane proxy.

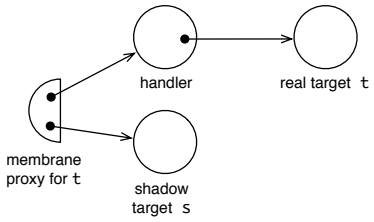

Fig. 6. A membrane proxy with a shadow target.

The purpose of the shadow target is to store wrapped properties of a frozen target. This way, when the handler returns a wrapper for the original target property from the `get` trap, the proxy checks the result against the shadow target, not the original target. Since the shadow target only contains wrapped properties, the invariant check will succeed. Thus, the membrane now operates correctly on frozen records.

We will henceforth refer to the shadow target simply as "the shadow", and to the real target as "the target". Having introduced a shadow, we now have two records (the

shadow and the target) that can be either frozen or non-frozen. We must ensure that this "frozen state" of both records remains synchronized. Otherwise, consider a membrane wrapper with a non-frozen shadow but a frozen target: when a client asks whether such a wrapper is frozen, the wrapper cannot answer `true`, as the proxy will check the answer against the state of the shadow, which is non-frozen.

We employ the following strategy to keep the shadow and the target in sync: as long as the real target is not frozen, the shadow is also not frozen. In this case, no invariants on the proxy are enforced, and the proxy is free to return non-identical, wrapped values from its `get` trap. There is no need to store these wrapped values on the shadow.

The first time that a proxy must reveal to a client that it is frozen[7], the proxy first *synchronizes* the state of the shadow with that of the target. It does this by defining a wrapped property on the shadow for each property on the target, and then freezing the shadow. Once the shadow is frozen, every time a property is accessed, the value returned is the wrapper defined on the shadow, not the original value.

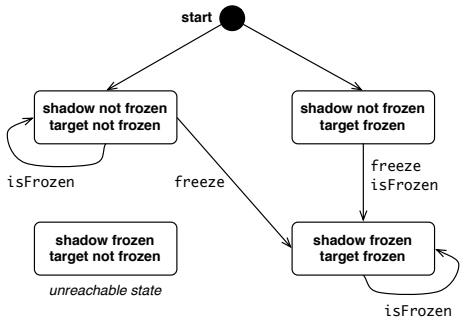

Fig. 7. State chart depicting the states of a membrane wrapper with a shadow target.

Figure 7 shows a state chart with the allowable states of a membrane proxy with a shadow target. The four possible states are determined by whether or not either shadow target or real target are frozen. A transition to the lower right state always implies a synchronization of the shadow target with the real target before freezing the shadow.

Figure 8 depicts the state of the proxy after synchronization with a target with a single `foo` property.

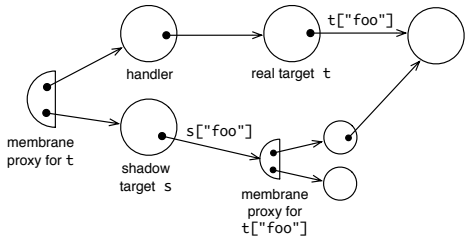

Fig. 8. After synchronization, the shadow caches wrapped values of the target.

[7] A proxy must reveal that it is frozen when intercepting `freeze` or when the proxy must answer `true` in response to an `isFrozen` test because the real target is frozen.

```
makeMembrane ≝ λx.
 var revoked = false;
   let wrap = λy.
    if typeof y = "constant"
       y
       if typeof y = "function"
           λz.assert (!revoked) ; wrap (y (wrap z))
           proxy {} {
            get : λt_s, s.assert (!revoked) ; if (isFrozen t_s) t_s[s] (wrap y[s])
            set : λt_s, s, v.assert (!revoked) ; y[s] := (wrap v)
            freeze : λt_s.assert (!revoked) ; freeze y ; (sync t_s y)
            isFrozen : λt_s.assert (!revoked) ; if (isFrozen y) (sync t_s y ; true) false
            enum : λt_s.assert (!revoked) ; keys y } ;
   {ref : wrap x
    revoke : λ.revoked := true}

sync ≝ λt_s, t_r.if (!isFrozen t_s)
                    (for (s : t_r) t_s[s] := wrap t_r[s] ; freeze t_s)
                    null
```

Fig. 9. Membranes with shadow target in λ_{TP}.

A membrane making use of this synchronization strategy between shadow and target is shown in Figure 9. Note that the first argument passed to **`proxy`** is an empty object $\{\}$ (the shadow) and not the real target y. Consequently, the first argument passed to each trap t_s denotes the shadow target. In the definition of "sync", t_s similarly stands for the shadow target while t_r stands for the real target.

To conclude this Section, we have shown how membranes can be interposed between two object graphs while preserving the frozen state of objects across both sides of the membrane *and* that operate correctly on frozen objects. The key idea is for a membrane proxy to not wrap the real target object directly, but rather to wrap a shadow target that can store wrapped properties of the target object. If the proxy is itself asked to become frozen, or to reveal that it is frozen via the `isFrozen` operator, it first synchronizes the state of its shadow before proceeding. This ensures that no invariant enforcement assertions will fail on the proxy.

4.4 Identity-preserving Membranes

In the previous Sections, we presented minimal yet useful implementations of the membrane pattern. Nonetheless, these implementations still have a number of issues regarding object identity that practical membrane implementations can and should address.

First, for non-frozen records, the wrappers are not cached, so if a record is passed through the same membrane twice, clients will receive two distinct wrappers, even though both wrap the same record. Consider the following λ_{TP} expression:

$$\texttt{let}\ x = \text{makeMembrane}(\{s : \{\}\}).\text{ref}\ ;\ (x[s] = x[s])$$

This expression will always reduce to `false` since each occurrence of $x[s]$ reduces to a fresh wrapper for the value of the s property.

Second, no distinction is made between the opposite directions in which a record can cross a membrane. If a record is passed through the membrane in one direction, and then passed through the membrane again in the opposite direction, one would expect to retrieve the original object. However, consider the following λ_{TP} expression, where x denotes a membraned proxy and v denotes a record:

$$x[s] := v\ ;\ \texttt{let}\ y = x[s]\ ;\ (v = y)$$

This expression will also always reduce to `false` since y will be a wrapper for a wrapper for v. In $x[s] := v$, v is wrapped when it crosses the membrane inwards. Then, in $y = x[s]$, the wrapped value is wrapped again when it crosses the membrane outwards. It would be better for these crossings to cancel each other out instead.

These limitations can be addressed by having the membrane maintain extra mappings that map proxies to their wrapped values and vice versa. The details are beyond the scope of this paper. Suffice it to say that in practice the above two problems are addressable in ECMAScript using WeakMaps[8], which are identity hashmaps with the weak referencing semantics of Ephemerons [13].

5 Discussion

The cost of transparent invariants Membranes, while being a very generic abstraction, are a useful case study because they aim to *transparently* interpose between two object graphs. For the membrane to be adequately transparent, it must be able to accurately uphold invariants across the membrane.

The invariant enforcement mechanism complicates membranes, as it prevents a membrane proxy from directly exposing the properties of a frozen target object as wrapped properties. Instead, we introduced a shadow target to hold the wrapped properties. This has two costs: first, there is the memory overhead of duplicating all the properties on the shadow. Second, operations that modify or expose the state of an object (i.e. `freeze` and `isFrozen`) must explicitly synchronize the state of the shadow and the real target.

While these overheads are not to be underestimated, we believe they are manageable in practice. First, we expect the dominant operations on objects to be `get` and `set`, not `freeze` and `isFrozen`. Second, wrapped properties are only defined on the shadow lazily, i.e. only when the proxy is about to reveal that it is frozen for the first time. If a proxy is never tested for frozenness, the shadow is never even used. Only when a proxy is revealed as frozen must transitive wrappers for the target properties be defined on the shadow.

[8] See `http://wiki.ecmascript.org/doku.php?id=harmony:weak_maps`.

Garbage collection In Section 4.1 we introduced revocable references. One of the primary use cases of such references is to facilitate memory management by reducing the risk of memory leaks. The idea is that if objects only hold a revocable reference to a certain resource object, then revoking that reference instantly removes all live references to the resource, allowing it to be garbage collected.

In our earlier Proxy API [7], a proxy did not store an implicit reference to a target object. Rather, it was the handler's responsibility to explicitly manage the reference to the target object, as shown in the first code snippet in Section 2.2. Upon revoking the proxy, that reference would be nulled out, allowing the garbage collector to collect the target object.

In the trustworthy Proxy API introduced here, the proxy holds an implicit reference to the target object which is not under programmer control. The proxy needs this reference to perform its invariant checks. Unfortunately this also implies that there is no way for the handler to null out this reference when the proxy is revoked. Hence, the revocable references introduced in Section 4.1 cannot be used for memory management purposes.

In Javascript, we solved this problem by providing revocable proxies as a primitive abstraction. This allows the proxy implementation to null out the references to its target and handler upon revocation.

Handlers and targets as proxies In Javascript, as in λ_{TP}, both the target and the handler of a proxy can themselves be proxies. The fact that a handler can itself be a proxy is a property that we have found useful in writing highly generic proxy abstractions. For a concrete example, we refer to our prior work [7].

The fact that the target of a proxy can itself be a proxy raises questions about the validity of our invariant enforcement mechanism. If the target is itself a proxy, might it not be able to return inconsistent results so as to mislead the invariant checks? After all, invariant enforcement hinges on the fact that the target object cannot lie about what invariants it upholds. Fortunately, this is not the case. We sketch an informal proof by induction on the target of a proxy.

First, we note that any chain of proxy-target links must be finite and non-cyclic: 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.

Alternatives to runtime assertions To make proxies trustworthy, we currently rely on run-time post-condition assertions on the return value of trap functions. Some of these assertions, most notably those for the `keys` trap, are relatively expensive, which has prompted us to look into alternative designs for achieving trustworthy proxies.

One design alternative (proposed to us by E. Dean Tribble) is to *ignore* the return value of trap functions altogether, and instead always forward the intercepted operation to the target object *after* having invoked the trap. The outcome of the operation on the proxy is then guaranteed to be the same as the outcome of the operation on the target, so invariants are preserved. This essentially turns traps into callbacks (event handlers) that get notified right before performing the operation on the target. Since the notification takes place *before* forwarding the operation, the trap may still indirectly determine the outcome of the intercepted operation by manipulating the target object.

While this design avoids run-time invariant checks, it has the downside of making it even harder to express virtual object abstractions, as the virtual object is forced to define a concrete property on the target object for every virtual property that is accessed. As we have not yet fully explored this design alternative, we will refrain from going into more detail here. It does teach us that our proposed design is not the only way of achieving trustworthy proxies, and that there is a broader design space with trade-offs to explore.

6 Availability

Trustworthy proxies are known as "direct proxies" in Javascript. There currently exist two implementations of direct proxies. The first is a native implementation in Firefox 18. The second is a self-hosted implementation via the `reflect.js` library[9]. `reflect.js` is a small Javascript library implemented by the first author. The library implements trustworthy proxies on top of our previously proposed Proxy API for Javascript [7] which is available natively in Firefox and Chrome. The library essentially uses untrustworthy proxies to implement trustworthy proxies in Javascript itself. An implementation of membranes that preserve invariants is shipped with the library.

7 Related Work

For an overview of related Proxy and reflection APIs, we refer to our earlier work [7]. Here, we specifically discuss related work on invariant enforcement in Proxy APIs.

7.1 Chaperones and Impersonators

Chaperones and impersonators are a recent addition to Racket [14]. They are the run-time infrastructure for Racket's contract system on higher-order, stateful values.

Chaperones and impersonators are both kinds of proxies. One difference is that impersonators can only wrap *mutable* data types, while chaperones can wrap both mutable and immutable data types. A second difference is that chaperones can only further *constrain* the behavior of the value that it wraps. When a chaperone intercepts an operation, it must either raise an exception, return the same result that the wrapped target would return, or return a chaperone for the original result. Impersonators, on the other hand, are free to change the value returned from intercepted operations.

[9] See `http://github.com/tvcutsem/harmony-reflect`.

Chaperones are similar to trustworthy proxies, in that they restrict the behavior of a wrapper. A trustworthy proxy that wraps a frozen object is constrained like a chaperone. It is actually more constrained, since a chaperone is allowed to return a chaperone for the original value, while trustworthy proxies are not allowed to return a proxy for the value of a frozen property. Conversely, as long as the wrapped object is non-frozen, trustworthy proxies are like impersonators and may modify the result of operations.

There are important differences between chaperones and trustworthy proxies, however. First, as chaperones are allowed to return *wrappers* for the original values, *even* for "immutable" data structures, they avoid the overhead of the shadow target technique that we employed for membranes. However, this comes at the cost of weakening the meaning of "immutable": accessing the elements of an immutable vector, wrapped with a chaperone, may yield new wrappers each time an element is accessed. Behaviorally, the element will be the same (modulo exceptions), but structurally it may have a different identity. Thus, the invariant that immutable vectors must always return an *identical* value upon access is weakened.

Second, trustworthy proxies may *both* be a chaperone and an impersonator at the same time. In Racket, a value can be classified as (permanently) mutable or immutable. This distinction cannot be made in Javascript: not only can objects be "half-mutable" (cf. the "partially configurable" state in Figure 1), their mutability constraints can also change at runtime (e.g. by calling `Object.freeze`). Hence, upon wrapping a Javascript object, one cannot decide at that time whether to wrap it with a chaperone or an impersonator. That is why trustworthy proxies must use dynamic checks to test whether to behave as an impersonator (no invariant checks required) or as a chaperone-like proxy (with restricted behavior).

7.2 Virtual Values

Starting from our initial Proxy API [7], Austin et al. have recently introduced a complementary Proxy API for virtualizing primitive *values* [2]. 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 trustworthy proxies, virtual values are proxies with a separate handler. The handler for a virtual value proxy 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 an index into another record.

An important difference between the trustworthy proxies API as presented here, and the virtual values API, is that the latter provides a general `isProxy` primitive that tests whether or not a value is a proxy. Our API does not introduce such an explicit test because it breaks transparent virtualization. As a design decision, we do not want clients to know or care that they are dealing with proxies for other objects.

For virtual values, the `isProxy` primitive was added to enable clients to defend themselves against malicious virtual values, such as mutable strings in a language that otherwise only has immutable strings. The idea is for clients to defend themselves against malicious proxies by explicitly testing whether or not the the object they are interacting with is a proxy. Virtual value proxies have no invariant enforcement.

Trustworthy proxies provide an alternative solution. As trustworthy proxies cannot violate language invariants, the better way for clients to protect themselves against erratic object behavior is to test whether an object is frozen, rather than testing whether it is a proxy.

7.3 Java Proxies

The `java.lang.reflect.Proxy` API [15], introduced in Java 1.3, enables one to intercept method invocations on instances of interface types. Like the Proxy API sketched here, a Java proxy has an associated handler object (known as an `InvocationHandler`) to trap invocations. Unlike trustworthy proxies, Java proxies do not necessarily wrap a target object.

The Java Proxy API only supports proxies for interface types, not class types. As a result, proxies cannot be used in situations where code is typed using class types rather than interface types, limiting their general applicability. Eugster [1] describes an extension of Java proxies that works uniformly with instances of non-interface classes, which, in addition to method invocation, can also trap field access and assignment.

Java proxies have little need for elaborate invariant enforcement. This is partly because Java provides no operations that change the structure of objects (the fields and methods of an object are fixed), and partly because Java proxies do not virtualize field access, so the notion of virtualizing a `final` field does not arise.

However, there is one invariant on Java proxies that is maintained via a runtime check: the runtime type of the return value of the `InvocationHandler`'s `invoke` method must be compatible with the statically declared return type of the intercepted method. If the handler violates this invariant, the proxy implementation throws a `ClassCastException`. This makes Java proxies trustworthy when it comes to the return type of intercepted method invocations.

8 Conclusion

Proxies are a useful part of reflection APIs that enable a variety of use cases, from generic wrapper abstractions such as membranes and higher-order contracts, to virtual object abstractions such as remote object references and lazy initialization. In a language with both proxies and language invariants, these features interact. If a proxy is allowed to emulate an object with language invariants, the question arises whether these invariants are still enforced by the language.

We presented *trustworthy* proxies, which are proxies that can wrap objects with (universal and monotonic) language invariants, where these invariants are enforced through runtime checks by the proxy mechanism. This ensures that proxies cannot circumvent these invariants, such that developers and the VM itself can continue to rely on these invariants even in the presence of proxies.

We explored the need for trustworthy proxies in the context of Javascript, and presented a formal semantics for λ_{TP}, an extension of the λ-calculus including trustworthy proxies. We have shown how abstractions such as transitively revocable references (i.e.

membranes) can be built using trustworthy proxies, achieving a transparent interposition between two object graphs which accurately represents language invariants on both sides of the membrane.

Acknowledgements We thank the members of the ECMA TC-39 committee and the `es-discuss` community for their detailed feedback on this work.

References

1. Eugster, P.: Uniform proxies for java. In: OOPSLA '06: Proceedings of the 21st annual conference on Object-oriented programming systems, languages, and applications, NY, USA, ACM (2006) 139–152
2. Austin, T.H., Disney, T., Flanagan, C.: Virtual values for language extension. In: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications. OOPSLA '11, New York, NY, USA, ACM (2011) 921–938
3. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. ICFP '02, New York, NY, USA, ACM (2002) 48–59
4. Pratikakis, P., Spacco, J., Hicks, M.: Transparent proxies for java futures. In: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. OOPSLA '04, New York, NY, USA, ACM (2004) 206–223
5. ECMA International: ECMA-262: ECMAScript Language Specification. Fifth edn. ECMA, Geneva, Switzerland (December 2009)
6. Fähndrich, M., Leino, K.R.M.: Heap monotonic typestates. In: International Workshop on Aliasing, Confinement and Ownership (IWACO '03). (2003) 58–72
7. Van Cutsem, T., Miller, M.S.: Proxies: design principles for robust object-oriented intercession APIs. In: Proceedings of the 6th symposium on Dynamic languages. DLS '10, ACM (2010) 59–72
8. Crockford, D.: Javascript: The Good Parts. O'Reilly (2008)
9. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of javascript. In: Proceedings of the 24th European conference on Object-oriented programming. ECOOP'10, Berlin, Heidelberg, Springer-Verlag (2010) 126–150
10. Redell, D.D.: Naming and Protection in Extensible Operating Systems. PhD thesis, Department of Computer Science, University of California at Berkeley (Nov 1974)
11. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, John Hopkins University, Baltimore, Maryland, USA (May 2006)
12. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja: Safe active content in sanitized javascript (June 2008) `tinyurl.com/caja-spec`.
13. Hayes, B.: Ephemerons: a new finalization mechanism. In: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. OOPSLA '97, New York, NY, USA, ACM (1997) 176–183
14. Strickland, T.S., Tobin-Hochstadt, S., Findler, R.B., Flatt, M.: Chaperones and impersonators: run-time support for reasonable interposition. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. OOPSLA '12, New York, NY, USA, ACM (2012) 943–962
15. Blosser, J.: Explore the Dynamic Proxy API. (2000) `http://www.javaworld.com/jw-11-2000/jw-1110-proxy.html`.