Property Trees: An Efficient Object Property Lookup Mechanism for Dynamic Languages

Brendan Eich, Andreas Gal, Luke Wagner, David Mandelin, Gregor Wagner, Roy Frostig, Boris Zbarsky, Jason Orendorff, Mike Shaver, David Anderson, Nicholas Nethercote, Jeff Walden

Mozilla Corporation
{brendan,gal,lw,dmandelin,gwagner,froystig, bzbarsky, jorendorff,shaver,danderson,nnethercote,jwalden}@mozilla.com

Abstract

JavaScript is a dynamically typed programming language widely used in browser-based web applications. As with many dynamic languages, JavaScript objects are essentially associative arrays that lack static typing; object properties can be added and removed at runtime. JavaScript also provides a prototype-based inheritance mechanism to create complex object hierarchies. As a result, property lookups in JavaScript imply a potentially slow search of the objects along the prototype chain. We present property trees as an efficient technique to dynamically infer structural types for objects. We show that, using this inferred type information, we can speed up property accesses and common sequences of property additions by up to 2.2x in some benchmarks.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — *Runtime Environments, Optimization*.

General Terms Design, Experimentation, Measurement, Performance.

Keywords JavaScript, property tree, property cache, shapes.

1. Introduction

Dynamic languages such as JavaScript, Python, and Ruby are popular since they are expressive, accessible to non-experts, and make deployment as easy as distributing a source file. They are used for small scripts as well as for complex applications. JavaScript, for example, is the de facto standard for client-side web programming and is used for the application logic of browser-based productivity applications such as Google Mail, Google Docs and Zimbra Collaboration Suite. Dynamic languages often use a hash table-like object abstraction that holds a set of properties that can be added and removed dynamically.

Compilers for statically typed languages rely on type information to generate space-efficient object representations and fast property access code. Accessing a Java [14] object field, for example, is as simple as reading the property value using a compile-type computed offset relative to the object base. In a dynamically typed programming language such as JavaScript there is no notion of class and property access is typically much slower than the equivalent property access in a statically typed language.

In this paper we describe a new strategy for implementing dynamic language objects efficiently using a *property tree* and a *property cache*.

The Self language [9] pioneered the use of *property map sharing* to represent dynamic objects with less memory and *polymorphic inline caching* (PICs) to speed property access. Property map sharing means that the representation of the set of properties and their attributes is shared among all objects that have the same properties. The objects only need to store unique copies of the property values. Polymorphic inline caching means that after performing an initial property lookup, the implementation patches the code, inlining a fast path for the same property lookup that can be used on future executions. The fast path can be used as long as the object on which the lookup is performed has the same property map. If the property map changes, the inlined fast path must be invalidated and replaced by the initial slow path or a newly valid fast path. Together, property map sharing and PICs make dynamic objects much more efficient.

These techniques do have a few disadvantages and limitations. The main disadvantage of PICs is that they require patching the code at run time. This makes the implementation fairly complex. Also, in multithreaded systems, the code cannot be shared across threads without locking, which may be slow. PICs also require invalidation, which is often hard to get right and is also complex to implement, because the invalidation code needs to know about all the generated fast paths and how to patch them back. Another problem with patching is that it is expensive on most processors. This is a problem especially if there is a code expression that accesses properties using objects with many different object property maps on different executions ("megamorphic" property access expressions), because then the code must be patched frequently.

To avoid all of these problems, we introduced the *property cache*, which achieves mostly the same effect without requiring any patching. This technique fits in especially well with pure interpreters that have limited code generation facilities. Instead of patching the code, we insert an entry into a fast associative array structure. Instead of running patched fast paths, we do a lookup into the associative array, and on a cache hit find a simple description of the steps needed to read the property. The property cache is not particularly bothered by megamorphic property access expressions–it simply incurs a low hit rate, and incurs the cost of creating useless property cache entries, which is much faster than creating useless

PLDI '10 June 5-10, 2010, Toronto, Canada

code patches. The property cache can also be invalidated simply and quickly: Instead of patching code for objects that are invalidated, we simply update the property cache key of those objects to a new unique value, so future cache lookups will miss.

A limitation of previously described property map sharing systems is that property maps are represented with hash tables, which must be copied and updated in order to add a new property, which is expensive. Also, two objects that have *almost* identical property maps must have two separate property maps, which mitigates the benefits of sharing.

The *property tree* is our solution to the problems with property map sharing. Instead of using hash tables, we represent all the shared property maps in the entire system with a single tree. Each node is labeled with a single property. Each node represents a shared property map, where the properties in the map are all those found on the path to the root node, which represents the empty map. This structure gives more sharing than the hash table solution. The property tree also makes adding properties much faster–instead of copying and updating a hash table, we need only add a new node to the tree.

Together, these techniques produced up to a 2x improvement in total application performance on standard benchmarks in a production interpreter, compared to the previous naïve property implementation.

The contributions of this paper are:

- The property tree, which represents shared property maps using less memory and with faster updates than previously described implementations.
- The property cache, an alternative to polymorphic inline caching (PICs) that is simpler to implement and has a simpler, faster invalidation strategy
- Implementation and experimental evaluation of these techniques in a production interpreter.

The remainder of this paper is organized as follows. In Section 2 we give an overview of name lookup in JavaScript. Section 3 describes our property tree abstraction and Section 4 details our use of the property tree and shape numbers to implement an efficient name lookup cache for JavaScript. Related work is discussed in Section 5. In Section 6 we evaluate our approach by observing its hit rate and behavior for a set of synthetic benchmarks and real-world web application workloads. The paper ends with conclusions in Section 7.

2. JavaScript and Name Lookup

We now consider some of the relevant aspects of the JavaScript language and formalize the required behavior of property and variable access. We refer to the lookup required for these two related operations as *name lookup*. Existing formalisms for JavaScript [4, 23, 10] provide operational semantics that capture many key features of the language and even interactions with the browser. However, for name lookup, these formalisms use a more restrictive form of lookup than allowed by JavaScript in the existing and upcoming ECMAScript standards [15, 22]. In particular:

- Properties can be looked up using first class string values, not just elements of an *a priori* fixed set of identifiers.
- The runtime lookup mechanism must consider more than just an object's own state; it must consider the object's prototype chain as well as the dynamic contents of its lexical scope.

2.1 Property Lookup

Properties may be added to an object when the object is created. The two main ways to create an object in JavaScript are: an object-literal syntax

```
cow1 = { weight:100,
         fatten:function() { this.weight++ } };
```

which defines an object's properties using a list of `name:expr` pairs, and a constructor syntax

```
function Cow(w) {
    this.weight = w;
    this.fatten = function() { this.weight++ };
}
cow2 = new Cow(100);
```

which involves writing a *constructor function* and then using it to construct an object with the `new` operator.

Once an object is created, properties can be added and removed dynamically:

```
assertEq(cow2.horns, undefined);
cow2.horns = true;
assertEq(cow2.horns, true);
delete cow2.horns;
assertEq(cow2.horns, undefined);
```

(`undefined` is the value returned when a non-existent property is accessed.) In this way, an object's set of properties can vary over time and, in general, is not fixed for a given program point.

JavaScript supports a form of inheritance/delegation through prototyping. Specifically, if an object `a` has another object `b` as its prototype, a property lookup `a.x` will first look for property `x` in `a` and, failing to find it there, look for `x` in `b`, and so on.

The prototype association is primarily established through the use of constructor functions as below:

```
function Shape(c) { this.center = c; };
Shape.prototype = { color:"red" };
s = new Shape({x:0, y:0});
assertEq(s.color, "red");
```

Here, `prototype` is a special property of the `Shape` function object which is used to set the prototype of all objects subsequently created by `Shape`.

Updates made to a property on an object `a` are immediately visible to all other objects which have `a` as their prototype. For example:

```
Shape.prototype.color = "blue";
assertEq(s.color, "blue");
```

The last relevant feature of property access is the ability to define *getter* and *setter* functions for properties. For example, the following code uses a getter and setter function to derive an `area` property from the `side` property of a square.

```
sq = { side:4,
       get area() { return this.side * this.side },
       set area(a) { this.side = Math.sqrt(a) } };
assertEq(sq.area, 16);  // call getter
sq.area = 25;           // call setter
assertEq(sq.side, 5);
```

Together, a getter and setter constitute a property, and thus the pair of functions can be inherited like data properties.

With this background, we can formalize the relevant definitions. We use $\rightharpoonup_{fin}$ to denote a finite partial mapping and, for such a mapping f, $f(x) \downarrow$ to mean "f is defined on x". Additionally, definitions are given in the context of a single fixed program state; a full operational semantics with dynamic name lookup is beyond

$$
\begin{array}{rcl}
s \in String & & \\
f \in Function & & \\
v \in Value & ::= & Object \cup String \cup \ldots \\
p \in Property & ::= & v \\
& | & \texttt{<}f_1\texttt{,}f_2\texttt{>} \\
pm \in PropMap & ::= & String \rightharpoonup_{fin} Property \\
mo \in Maybe(Object) & ::= & Object \cup \{null\} \\
mp \in Maybe(Property) & ::= & Property \cup \{null\} \\
o \in Object & ::= & \{pm\texttt{,}mo\}
\end{array}
$$

Figure 1. JavaScript object

$$
prop : Object \times String \rightarrow Maybe(Property)
$$

$$
prop(\{pm\texttt{,}mo\}, s) = \begin{cases} pm(s) & \text{if } pm(s)\downarrow, \\ prop(mo, s) & \text{if } mo \neq null, \\ null & \text{otherwise.} \end{cases}
$$

Figure 2. Property lookup

the scope of this paper. Hence, the standard indirection through a domain of heap addresses is not necessary and, for simplicity, the $Object$ and $Property$ domains are used directly.

Figure 1 defines the components of a JavaScript object necessary for name lookup. Figure 2 defines the effect-free portion of property lookup (i.e., up to a potential getter or setter call). Having performed the lookup $prop$, the interpreter can take the appropriate action by changing the data value, calling the getter/setter, or returning `undefined`.

2.2 Variable Lookup

In JavaScript, names that are not explicitly accessed as properties (using the syntactic form "`e.o`") perform a lookup in the current scope. While scopes are nested lexically, their contents are dynamic and, in general, identifiers cannot be resolved to variables at compile time. One source of dynamism is a special object, called the *global object*, which forms the root of every scope chain and whose properties are treated as global variables. Hence, after executing the script:

```
x = 0;
function f() { var y; y = 0; z = 0; };
f();
```

the global object has properties named "`x`" and "`z`". As an object, the global object can have its properties dynamically added and removed:

```
function g() { return x; }
x = 10;
assertEq(g(), 10);
delete x;
g();      // ReferenceError: x is not defined
```

Objects can also be inserted further down the scope chain using the `with` operator. A block `with(o) { ... }` uses object `o` as a nested local scope of code in the block, with `o`'s properties serving as local variables. For example:

```
function h(o) {
  var x = 42;
  function h() { with(o) { return x; } }
  return h();
}
assertEq(h( {/* empty object */} ), 42);
assertEq(h( { x:"moose" } ), "moose");
```

$$
\begin{array}{rcl}
vm \in VarMap & ::= & String \rightharpoonup_{fin} Value \\
sc \in Scope & ::= & \texttt{call(}vm\texttt{,}sc\texttt{)} \\
& | & \texttt{with(}o\texttt{,}sc\texttt{)} \\
& | & \texttt{global(}o\texttt{)}
\end{array}
$$

Figure 3. JavaScript scope

$$
var : Scope \times String \rightarrow Maybe(Property)
$$

$$
\begin{array}{rcl}
var(\texttt{call(}vm\texttt{,}sc\texttt{)}, s) & = & \begin{cases} vm(s) & \text{if } vm(s)\downarrow, \\ var(sc, s) & \text{otherwise.} \end{cases} \\
var(\texttt{with(}o\texttt{,}sc\texttt{)}, s) & = & \begin{cases} prop(o, s) & \text{if } prop(o, s)\downarrow, \\ var(sc, s) & \text{otherwise.} \end{cases} \\
var(\texttt{global(}o\texttt{)}, s) & = & prop(o, s)
\end{array}
$$

Figure 4. Scope lookup

Lastly, even the contents of local scopes can change at runtime, due to the fact that `eval` executes in the current scope:

```
var g = 42;
function i(s) {
  eval(s);
  return g;
}
assertEq(i(""), 42);
assertEq(i("var g = true"), true);
```

To give a formal definition of variable lookup, we first give a recursive definition of $Scope$ in Figure 3 that handles the cases described above: local scopes of nested calls, `with` blocks, and the global object. Based on this, Figure 4 defines variable lookup.

Another concept, used in subsequent sections, that can be defined formally in terms of the given definitions is *shadowing*. A property is shadowed, in the context of a particular lookup, if another property with the same name is found first by the lookup function. This is captured by the predicate $shadow$ in Figure 5 which returns whether the given identifier names a shadowed property for the given object or scope.

3. Property Tree

A naïve implementation of JavaScript objects[1] could give each object a hash table mapping names to properties and a pointer to the prototype. But in many programs, this strategy wastes memory storing redundant property maps. Consider this example constructor function:

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
```

[1] JavaScript scopes are similar to objects except that most scopes do not have a prototype, so we will simply refer to *objects* in our presentation.

$$shadow : Object \times String \rightarrow Bool$$

$$shadow(\{pm, mo\}, s) = \begin{cases} false & \text{if } mo = null, \\ prop(mo, s)\downarrow & \text{else if } pm(s)\downarrow, \\ shadow(mo, s) & \text{otherwise.} \end{cases}$$

$$shadow : Scope \times String \rightarrow Bool$$

$$\begin{aligned} shadow(\texttt{call}(vm, sc), s) &= vm(s)\downarrow \wedge var(sc, s)\downarrow \\ shadow(\texttt{with}(o, sc), s) &= shadow(o, s) \vee \\ &\quad prop(o, s)\downarrow \wedge var(sc, s)\downarrow \\ shadow(\texttt{global}(o), s) &= false \end{aligned}$$

Figure 5. Property shadowing

If the program creates one million `Point` objects, the naïve implementation creates one million identical maps giving the names, getters, setters, attributes, and storage location of `x` and `y`. In the SpiderMonkey JavaScript VM [1], for example, small objects are 32 bytes, and property definitions are 20 bytes.[2] Thus, a program that creates one million `Points` uses 32 MB for the objects, and at least 40 MB for redundant property maps, so over half the allocated space is wasted.

The creators of Self noticed this problem and solved it by giving each object a pointer to the property map and sharing the maps among objects [9]. This reduces the memory required for property maps in our example to only 40 bytes or so.

The basic version of the shared property map technique uses a hash table for each shared map. Initially, objects have the same map as their prototype, which is how the sharing is achieved in practice. If the property map of an object ever needs to be modified, the shared map is copied, and then the copy is modified. This strategy has two inefficiencies. First, objects with slightly different property maps must use distinct hash tables, and do not share their redundant information. Second, hash tables have to be copied when properties are added or modified, which is a relatively expensive operation.

Our technique solves both of these problems by representing property maps in a tree that allows partially redundant maps to overlap, and can generate new scopes without copying. In the rest of this section, we explain the property tree in detail and analyze its memory usage quantitatively.

Given P independent, non-unique properties each of size S words mapped by all scopes in a runtime, we construct a property tree of N nodes each of size $S + L$ words (with L words being used for tree linkage). A nominal L value is 2 for leftmost-child and right-sibling links. We hope that $N < P$ by enough that the space overhead of L, and the overhead of scope entries pointing at property tree nodes, is worthwhile.

3.1 Property tree construction

The property tree is constructed as follows. If any empty scope S in the runtime has a property x added to it, find or create a node under the tree root labeled x, and set $S_{LastProp}$to point at that node.

If any non-empty scope S whose $S_{LastProp}$is labeled y has another property z added, find or create a property tree node labelled z under $S_{LastProp}$, and set $S_{LastProp}$ to point at that node. This produces the following path to root in the property tree:

$$* \leftarrow x \leftarrow \ldots \leftarrow y \leftarrow z \leftarrow S_{LastProp}$$

[2] Properties are actually 32 bytes, but 12 of those are for the optimizations described in this paper and would not be used in a naïve implementation.

A property is labeled by its members' values: the property name, getter function, setter function, slot-storage offset, property attributes (e.g. *read-only*), short id (used for quick lookup of frequently encountered property names such as *length*), and a field indicating iteration order.

Sidebar on iteration order: The ECMAScript Specification [15] requires no particular order, but traditionally web-compatible JavaScript engines have promised and delivered property definition order and web authors have come to rely on it. We could use an order number per property, which would require a sort when enumeration commences, and an entry order generation number per scope. An order number beats a list, which should be doubly-linked for $O(1)$ delete. An even better scheme is to use a parent link in the property tree, so that the ancestor line can be iterated from $S_{LastProp}$ when creating the list of keys to enumerate. This parent link also helps the garbage collector to sweep properties iteratively.

Note that labels are not unique in the tree, but they are unique among a node's children (barring rare and benign multi-threaded race condition outcomes, see below) and along any ancestor line from the tree root to a given leaf node.

Thus the root of the tree represents all empty scopes, and the first ply of the tree represents all scopes containing one property, etc. Each node in the tree can stand for any number of scopes having the same ordered set of properties, where that property represented by the node was the last added to the scope.

3.2 Property Deletion

What if a property y is deleted from a scope? If y is the last property in the scope, and its predecessor is x, we simply adjust $S_{LastProp}$ to x after we remove the scope's value table entry corresponding to that property node. The parent link mentioned in the iteration sidebar above makes this adjustment $O(1)$.

$$\ldots \leftarrow x \leftarrow y$$

If the deleted property y comes between x and z in the scope, modeling the deletion appears to require "forking" the tree at x, leaving $x \leftarrow y \leftarrow z$ in case other scopes have those properties added in that order; and to finish the fork, we have add a node labeled z with the path $x \leftarrow z$, if it doesn't exist. In pathological cases this can result in the creation of a large number of extra nodes, and to $O(n^2)$ growth when deleting many properties.

To avoid such pathological cases and retain predictable $O(1)$ behavior, we disengage scopes from the property tree regimen when the first property is deleted from a position other than $S_{LastProp}$. For such scopes we fall back onto the traditional hash-table implementation. Web code rarely makes use of property deletion, making this a rare compensation path. We measured about 0.09% property deletion rate per scope.

3.3 Thread Safety

While web JavaScript has "run to completion" semantics for individual JS programs, specific embeddings of JavaScript might prefer the concurrent execution of JavaScript programs sharing a common heap (and thus a shared property tree).

If the property tree operations done by concurrent threads are *find-node* and *insert-node*, then the only concurrency hazard is duplicate insertion. This is harmless except for minor bloat.

When all concurrent thread accesses to the property tree have ended or been suspended, the garbage collector is free to sweep the tree after marking all nodes reachable from scopes, performing *remove-node* operations as needed.

3.4 Efficiency

To decide whether the property tree is worthwhile compared to direct property storage in each scope, we must find the relation $\diamond$

between the number of words used with a property tree and the number of words required without a tree.

We model all scopes as one super-scope of capacity T entries (with T being a power of 2). Let α be the load factor of this double hash table. With the property tree, each entry in the table is a word-sized pointer to a node that can be shared by many scopes. But all such pointers are overhead compared to the situation without the property tree, where the table stores property nodes directly, as entries each of size S words.

With the property tree, we need $L = 2$ extra words per node for sibling and child pointers. Without the tree, $(1 - \alpha) * S * T$ words are wasted on free or removed sentinel-entries required by double hashing.

Therefore,

$$\begin{aligned} (\textit{property tree}) &\diamond (\textit{no property tree}) \\ N*(S+L)+T &\diamond S*T \\ N*(S+L)+T &\diamond P*S+(1-\alpha)*S*T \\ N*(S+L)+\alpha*T+(1-\alpha)*T &\diamond P*S+(1-\alpha)*S*T \end{aligned}$$

Note that P is $\alpha * T$ by definition, so

$$N*(S+L)+P+(1-\alpha)*T \diamond P*S+(1-\alpha)*S*T$$

which we can rewrite as follows:

$$\begin{aligned} N*(S+L) &\diamond P*S-P+(1-\alpha)*S*T-(1-\alpha)*T \\ N*(S+L) &\diamond (P+(1-\alpha)*T)*(S-1) \\ N*(S+L) &\diamond (P+(1-\alpha)*P/\alpha)*(S-1) \\ N*(S+L) &\diamond P*(1/\alpha)*(S-1) \end{aligned}$$

Let $N = P * \beta$ for a compression ratio β, $\beta <= 1$:

$$\begin{aligned} P*\beta*(S+L) &\diamond P*(1/\alpha)*(S-1) \\ \beta*(S+L) &\diamond (S-1)/\alpha \\ \beta &\diamond (S-1)/((S+L)*\alpha) \end{aligned}$$

For $S = 6$ (32-bit architectures) and $L = 2$, the property tree wins iff

$$\beta < 5/(8*\alpha)$$

We ensure that $\alpha \leq 0.75$, so the property tree wins if $\beta < 0.83$. An average β for browser startups is approximately 0.6.

3.5 Compressed Child Table

So far we have assumed a binary tree with $L = 2$. However, in many cases we observe that the property tree degenerates into a list of lists if at most one property Y follows X in all scopes. In or near such a case, we waste a word on the right-sibling link outside of the root ply of the tree. Note also that the root ply tends to be large, so $O(n^2)$ growth searching it is likely, indicating the need for hashing (but with increased thread safety costs).

If only K out of N nodes in the property tree have more than one child, we could eliminate the sibling link and overlay a children list or hash-table pointer on the leftmost-child link (which would then be either null or an only-child link; the overlay could be tagged in the low bit of the pointer, or flagged elsewhere in the property tree node, although such a flag must not be considered when comparing node labels during tree search).

For such a system,

$$L = 1 + (K * averageChildrenTableSize)/N$$

instead of 2. If $K << N$, L approaches 1 and the property tree is worthwhile if $\beta < .95$.

We observe that fan-out below the root ply of the property tree appears to have extremely low degree (Figure 6), so instead of a hash-table we use a linked list of child node pointer arrays.

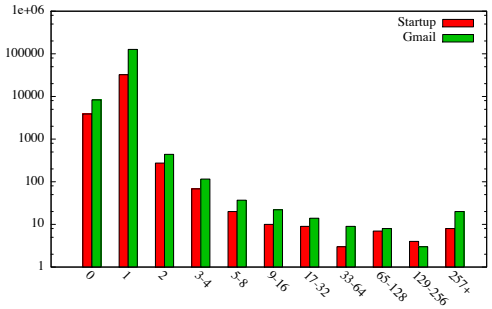

Figure 6. Histogram of child nodes in the property tree. Most property tree nodes have no or one child node.

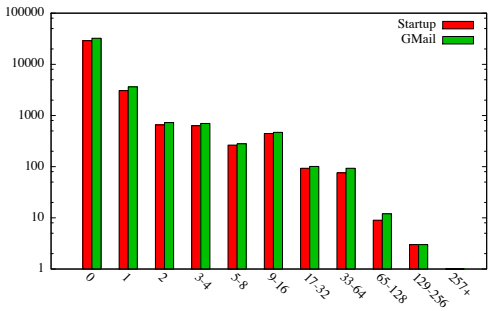

Figure 7. Histogram of properties per scope. The mean number of properties per scope is relatively small ($N < 5$) with a large standard deviation $N_\sigma \approx 8$.

3.6 Property Lookup

So far we have assumed that each scope contains a table with a pointer into the property tree for each property. Figure 7 shows that the mean number of entries per scope is relatively small ($N < 5$), with a large standard deviation $N_\sigma \approx 8$.

Instead of always allocating a table for every scope, we leave it null while initializing all the other scope members as if it were non-null and minimal-length. Until a property is added that crosses a threshold of 6 or more entries for hashing (or until a "middle delete" occurs and the scope opts out of the property tree regimen), we use linear search from $S_{LastProp}$ to find a given property, and save on the space overhead of a hash-table.

4. Property Cache

Property access in dynamic languages with prototypes is complex and potentially expensive. Even without prototypes, a simple lookup requires searching the property tree for the property. After finding the property, the implementation runs the operations specified by the property to get or set the value. With prototypes and nested lexical scopes, a lookup requires searching the property tree for each property in the prototype or scope chain until the property is found or the end of the chain is reached. The result is that even in the simplest case, property lookup requires traversing pointer structures and is tens or hundreds of times slower than in a language like Java. Also, prototypes and nested lexical scopes incur a high runtime cost.

Consider this code example:

```
function init(points) {
    for (var i = 0; i < 10; ++i)
```

```
        points[i].x = get_x(i);
}
```

The variables `i` and `points` can be bound statically, but reading the *i*th array element, the `x` property, and the `get_x` function must be done dynamically. Notice that if every element of `points` is a `Point` object, the property, and hence the property access operations, for `.x` are the same every time. Similarly, the property access operations for `get_x` may also be the same every time, and furthermore, the value of the property (the function itself) may also be the same every time. But reading the ith element is not the same every time: the property and its storage location are different on every iteration. Thus, we observe that for property accesses that use the dot syntax (in which the property name is given statically), the property operations are likely to be the same every time. This gives us the opportunity to speed up property access by caching the operations.

4.1 Overview

For the overview, we will consider the simple case of caching property accesses for single objects, with no prototype chain or lexical scope chain. The goal here is to avoid searching the property map in favor of a faster cache read.

In this simple case, the property access operations are the same for two property accesses if the objects have the same property maps and the property names are also the same. This suggests we design a cache where the key consists of the object property map and the property name. The value should be a simple description of how to access the property. For simple objects, properties are simply stored in an array, so the value would be simply the index into the array. For objects with getters and setters, the value would be a pointer to the getter and setter.

The strategy above would work, but there are some language and implementation considerations that lead us to use a somewhat different key. First, in a typical interpreter, the property name is an additional operand to the bytecode that would need to be loaded from memory to form the key. But note that the property name is fixed for a given interpreter PC (because we will only cache for dot-syntax operations), so we can use the interpreter PC for the key instead, saving the load.

Second, certain language features mean that objects can have identical property maps but should not be considered to have identical property access operations. For example, a JavaScript object can be *sealed*, which means that it is read-only, and an attempt to write to its properties throws an exception. Thus, we must avoid getting a cache hit for a sealed object when the cache entry was originally created for a non-sealed object. We can solve this problem by using a *shape number* in place of the property map for the key. For most objects, the shape number is simply a unique identifier for the property map. But when required, objects can be given their own unique shape number, so they will not get property cache hits from any other objects.

Thus, in our property cache:

- The **key** is a pair of the interpreter **PC** and object **shape number**.
- The **value** represents the property access operations. We have three kinds of values, distinguished by tag bits:
  - In the normal case, we have a simple property without getter or setter and the value is simply the array index of the property value.
  - For more complex cases, such as when there is a getter, the value is a pointer to the property definition.
  - As our example suggests, for function-valued properties, not only are the property access operations usually the same, but the value itself is usually the same. Thus, for function-valued properties, the cache value is the function value itself.

For speed, the cache is a fixed-size hash-table with no chaining. Collisions simply cause old entries to be overwritten.

Statically typed programming languages generate property access code that is valid for all objects of a specific *class*. As a dynamically typed language, JavaScript lacks the notion of classes.[3] A `Point` object can only be identified as such based on the observation that it has a certain set of properties (i.e. `x`, `y` and `color`).

4.2 Shapes

As we have discussed in Section 3, each object or scope embeds a pointer into the property tree pointing at the last property that was added. We could consider this pointer to be a *hidden class* [9, 3], and use it as part of each key in the property cache. All objects with a last property pointer pointing to the same property descriptor behave identically as far as a property lookup in the object itself is concerned. However, objects with identical direct properties (sometimes also called *own* properties) could have different prototypes, producing different overall lookup results. Therefore we would have to embed not only the last property pointer into each cache entry, but also the precise composition of the entire prototype chain to guarantee that the cached access is applied to the object in question.

Instead, we assign every object a machine-word sized shape number. Property cache entries use the shape field in their key, and for a property cache hit the shape number in the cache entry must match the object's shape. Shapes are assigned to objects in such a way that the following guarantees are maintained:

- **Basic layout guarantee** – If object x has shape s and later object y has the same shape s, then x and y have a set of properties with the same: names, getters, setters, attributes, slot-storage offsets, and ordering.
- **Sealed scope guarantee** – If object x has shape s and is not sealed and later object y has the same shape s, then y is also not sealed.

Notice the shape of an object does not cover anything about the object's prototype or parent or anything about the values of the object's own properties (beyond the method and branded scope guarantees we will discuss in the next section). In particular, a shape does not cover the types of property values: it is possible for $\{a : \text{“}x\text{”}\}$ and $\{a : 12\}$ to have the same shape.

4.3 Shape Evolution

We use an incremental approach to assign shapes that maintain the guarantees laid out in the previous section. Each property descriptor in the property tree contains a shape number. The root of the property tree represents all empty scopes, and is assigned the shape number 0. A strictly increasing shape number generator G is used to assign a unique shape number to all property descriptors in the property tree as they appear. Shape numbers are never re-used.

In JavaScript, all objects are initially created empty, which means that we initially assign them the shape number of the empty scope, which is 0. For a more compact representation, we split scopes and objects. Each object contains a pointer to a scope, which in turn stores the shape number of the last property that was added

[3] ECMAScript uses the term *class* to describe objects with specific behaviors, such as `Date` and `String` objects. This is not to be confused with classes in the type sense. A JavaScript `Date` class can still have an arbitrary number of properties and the fact that it was born as a `Date` object doesn't guarantee the presence or absence of any specific properties.

to it. As properties are added to or removed from the scope, we update the shape number in the scope. While not strictly necessary (we could look up the shape number in the property descriptor pointed to by $S_{LastProp}$ for scope S), this does avoid an extra indirection when performing shape checks on the scope.

By copying the pre-assigned shapes found in the property tree into the scope we obtain predictably evolving shape numbers for scopes that match the shape number of other scopes with the same properties (*basic layout guarantee*).

4.4 Unique Shapes

A scope may be given a unique shape that differs from the shape of the last property that was added to it. This is done in order to "bump" the shape of one scope without affecting other objects that happen to have the same properties. Such a "bump" is necessary to model special circumstances that should force invalidation of certain property cache entries (or code generated by the just-in-time compiler).

For example, when sealing an object, we must prevent any future property cache hits for property sets on that object. We do this by giving the sealed scope a unique shape. Existing property cache entries that cache access information for property-set operations become invalid since we precondition property cache hits on a matching shape; the resulting property cache miss will purge the existing property cache entry and re-fill it with new access information. In case of a sealed scope, the new cached access information instructs the VM to silently ignore the property set, as dictated by the language specification. By assigning unique shapes to sealed scopes we satisfy the *sealed scope guarantee*.

Note that two different objects may have the same "unique" shape. The shape uniquely identifies a point on a path of property operations, it does not necessarily identify a particular object.

4.5 Shadowing

As discussed in Figure 2.2, properties can be shadowed along the prototype and scope chains. Since shapes don't make any guarantees about the prototype relationship between objects, for property lookups that result in a hit along either the prototype or scope chains we would have to perform a multi-step lookup to ensure that each object along the chain has the same shape as at the time when we filled the property cache. If any of the shapes changed, the property we are looking for might have been shadowed by a newly defined property closer to the direct objected we started the search with.

This is particularly unfortunate considering that in most programs, usually only a small set of objects are actually used as prototype or parent objects (delegate objects). When invoking the `toString()` method on any string object, for example, the method is almost always found by walking along the prototype chain from a string object to *String.prototype*.[4] Walking up the prototype chain every time `toString()` is called only to find the same receiver object there is wasteful.

Lexical scopes behave similarly to object hierarchies. When we look up a global variable in a chain of scopes we often walk past a predictable path of scope objects before arriving at the global object where we find the property we are looking for. To be able to execute such non-direct property accesses (properties found along the property or scope chains) via property cache hits, we observe that a small set of objects serve as delegate objects, and add the following two guarantees to our shape rules:

- **Prototype chain shadowing guarantee** – If the property lookup $x.p$ finds property p in x's prototype chain on object y which has shape s, then if later y still has shape s, the lookup $x.p$ will still find p on y.

$$x \rightarrow \ldots \rightarrow y \rightarrow \ldots \rightarrow \textit{Object.prototype}$$

- **Scope chain shadowing guarantee** – If the variable lookup for p in a scope chain starting at sc finds p on a scope y of shape s, and there are no scopes before y along the scope chain that have prototypes, then if later y still has shape s, the lookup for p in sc will still find p on y.

$$\begin{array}{ll} sc & \\ \downarrow & \\ \ldots & \\ \downarrow & \\ \textit{global} & \rightarrow \ldots \rightarrow y \rightarrow \textit{Object.prototype} \\ & \\ \rightarrow & \textit{prototype chain} \\ \downarrow & \textit{scope chain} \end{array}$$

Using these shadowing guarantees, it is sufficient to first guard on the shape of the scope chain and, using information stored in the cache entry, the shape of the prototype object on which the property was found.

To ensure the shadowing guarantees above, we use a write barrier in the property set path. Every time a property is set, we test the property cache to see if we are about to shadow a property, in which case we walk from the direct object to the original delegate object and assign a unique shape to each scope along the way. This automatically invalidates any cached access information for the direct object or any intermediary objects along the prototype and scope chains.

4.6 Branded Scopes

Function-valued properties can be invoked like methods in JavaScript using the *object.method*() syntax. In practice, such method *values* very rarely change once assigned. We use this observation and brand scopes with a unique shape when a property in the scope is assigned a function value. This provides us with the following final guarantee:

- **Branded scope guarantee** If at time t_0 the object x has shape s_x; and x's scope is branded and x has an own property p, which is a non-readonly, Function-valued property; and at time t_1 an object y has shape s_x; and no shape-regenerating GC occurred; then y is x, and at time t_1 x's own property p has the same Function value it had at time t_0. (The existence of this property is guaranteed by the basic layout guarantee above.) (Informally: if x has a branded scope, changing any of x's own methods will change its shape.)

With this optimization we can cache the value of invoked functions in the property cache. As long the shape of the receiving object matches the shape in the property cache entry, we can directly proceed with the function call without even having to retrieve the value of the function from the object.

4.7 Polymorphic Lookups

Property accesses are not always monomorphic. Consider the following JavaScript code:

```
var x = [ { x:5, y:7, color:Red },
          { x:2, y:4, w:12, h:8, color:Blue } ];

function color(y) {
    return y.color;
}
```

[4] As we will discuss in a moment, method property lookup differs from regular property lookups due to scope *branding*. For the sake of this example, we will ignore branding momentarily.

List x contains `Point` and `Rectangle` objects. Both have a `color` property. Due to their differing structural types, *color* is stored in different object slots. Assuming an equally distributed access pattern, accessing the `color` property would result in constant cache misses and refills if we were to use the program counter as the sole property cache key. To be able to deal with such polymorphic property accesses we key the property cache using the program counter as well as the shape of the object the lookup starts with. In this concrete example thus we would consult the property cache with (pc, s_{Point}) for points and (pc, s_{Rect}) for rectangles, and each access variant would (likely) be mapped to a different cache location, avoiding unnecessary collisions and cache flushes.

4.8 Garbage Collection

To generate unique shapes we have to ensure that generated shape numbers are never reused. We don't keep track of wasted shapes that were used temporary for some scope with a unique shape that is no longer alive. Instead, occasionally during garbage collection we regenerate all shape numbers in order to recycle unused shape numbers so that we don't run out. This could end up in changing the shape of every object, so the property cache (and the just-in-time compiler code cache) are emptied when this happens.

Regenerating shapes follows the same rules as initial shape assignment. We apply the same numbering rules to re-number all property descriptors in the property tree, and then proceed to copy the new shapes into scopes referencing them and generate unique shape numbers for scopes as needed.

We carefully track the utilization of shape numbers as the program executes and schedule a shape-generating garbage collection as needed. For implementation reasons, our shape space only consists of 2^{24} distinct shapes. If there come to be more than 2^{24} objects that need distinct shapes, the shape rules can no longer be satisfied. The property cache cannot operate without them, so it is disabled permanently. In practice this condition is very unlikely to occur.

5. Related Work

JavaScript combines a number of aspects of different programming languages. It adapted the prototypal inheritance and dynamic objects from Self [9], closures from Scheme [13], lexical conventions from Java [14] and the regular expressions from Perl [2].

Self was an early, well-known prototype-based language. Self is based on Smalltalk [11] and shares many of Smalltalk's key concepts [20]. Chambers and al. [9] introduced a fast property lookup mechanism with the first implementation of Self. Self uses maps to represent shared members of a clone family. Each map contains slot names, pointers to slot code or constant slot values, slot-storage offsets in objects, and metadata indicating whether each slot is a parent slot. Self's hierarchy of maps is similar to our property tree.

The main difference between our work and Self's maps is our notion of *shapes* and the use of *unique* shapes to detect various hazards such as object sealing and property shadowing.

Self uses its maps to implement *polymorphic inline caching*, which is conceptually very similar to our property cache. The main difference between the approaches is that we use one shared cache whereas Self uses small dedicated caches per site. Our property cache can cache a much larger number of polymorphic cases per site, but has the disadvantage of possible collisions across sites.

Google's V8 JavaScript engine [3] implements *hidden classes*, which are conceptually very similar to our property tree. Just as Self V8 does not use unique shapes to invalidate cache entries in case of shadowing or object sealing.

Most prototype based languages share either a delegation or cloning based inheritance mechanism. Cecil [7] implements a prototype-based object model with multiple dispatch. Furthermore, it provides predicate classes [8] instead of slot-based dynamic inheritance.

Alternatively to dealing with the full flexibility of dynamic property lookups and prototype-based inheritance, existing work has suggested to constraint dynamic behavior to reduce lookup cost.

Dony et al. [12] present a survey about prototype based languages and explore their advantages and disadvantages. They propose a model where prototypes should be represented either with methods and variables or with slots but at least follow the implementation of an encapsulation mechanism. Delegation should be implicit and achieved by creating split objects.

Borning [6] compares classes versus prototypes in object-oriented languages. He uses constraints to establish and maintain inheritance relations. To overcome the slow property lookup, he suggests to automatically compile property lookup methods.

Steyaert et al. [19] combine class- and object-based Inheritance. They show the encapsulated inheritance on objects and combine the object model of class-based languages with the more orthogonal inheritance model of object-based inheritance.

Lieberman [17] studies the efficiency between delegation and inheritance approaches based on time/space tradeoffs. Inheritance based approaches may be more efficient because of fewer messages but the advantage of a delegation based solution is the smaller object size. Smaller objects make for faster object creation times, which can be important in systems that create large numbers of small objects with short lifetimes.

Moore presents Guru [18], an automatic restructuring and refactoring of hierarchy object models for Self. Guru takes a collection of objects and restructures them into a new inheritance hierarchy in which there are no duplicated methods and the behavior of objects is preserved.

A related problem to fast property lookup is effective dynamic method lookup.

André et al. [5] discuss the problem of huge memory consumption for static method lookup caches. They propose an incremental coloring algorithm that still guarantees the constant lookup cost and no expensive conflict graph generation. Similar work was done by Vitek et al. [21]. They show how to increase the efficiency of dynamic binding by cached inheritance search or selector-indexed table look-up.

Static analysis has also been used successfully to analyze and optimize property lookup behavior. Jang et al. [16] use points-to analysis for JavaScript to detect redundant property references.

6. Benchmarks

To evaluate our property tree and property cache implementation, we implemented them in the open source JavaScript VM SpiderMonkey [1], which is used by Mozilla Firefox. As a result of this choice we are able to provide benchmark numbers for in-browser synthetic benchmarks as well as actual JavaScript web applications.

All experiments were performed on a Mac Pro with 2 x 2.66 GHz Dual-Core Intel Xeon processor and 4 GB RAM running MacOS 10.6 and a version of Firefox 3.6beta3 that uses the property tree and property cache mechanisms we have introduced in this paper.

First we measured the cache hit ratio for the SunSpider Benchmark suite. As shown in Figure 8 we obtain an almost 100% hit rate for most benchmark programs. The property cache is clearly well suited for high regular and computationally intensive JavaScript code. Figure 9 shows the cache hit ratio for a number of web applications. Again computationally intensive JavaScript code display good cache hit rates. For less computationally intensive JavaScript code the hit ratio is substantially lower (70% for 280slides).

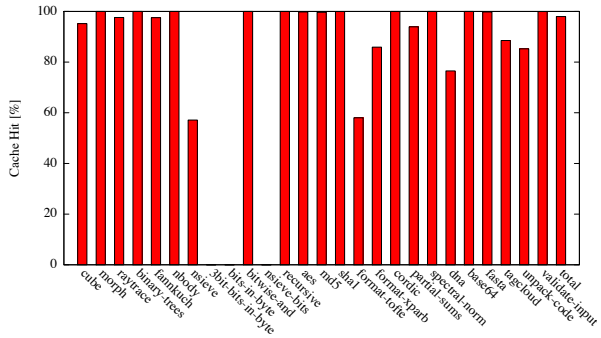

Figure 8. Cache Hit Ratios for the SunSpider Benchmark Suite. No results are reported for benchmark programs that have only a handful of name lookups ($N < 5$), which strongly biases the cache hit ratio (see Figure 13).

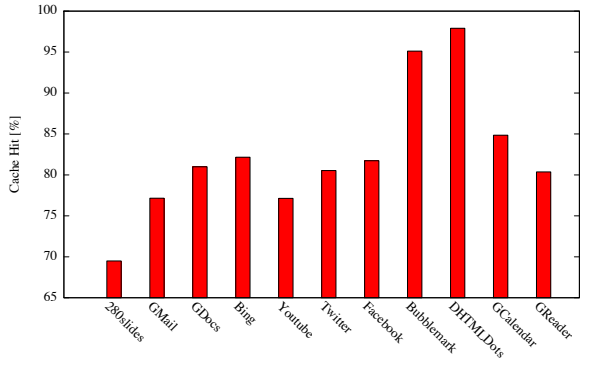

Figure 9. Cache Hit Ratios for a set of popular web applications. The most computationally intensive applications have the highest cache hit ratios ($r > 95\%$).

Cache misses are predominantly caused by collisions or initial fills (Figure 10). To reduce these misses we would have to increase the size of the property cache. We currently use 2^{12} property cache slots. Substantially increasing the size of the cache would almost inevitably cause poor machine L2/L1 cache behavior. For the Bubblemark benchmark we suffer from foreshadowing cache misses along the prototype chain due to the benchmark's particular use of a large number of objects of different shapes that are accessed from the same location. We are evaluating techniques to key such cases differently into the property cache to avoid constant cache refilling.

For most web applications such "deep prototype hits" are not particular relevant. The overwhelming number of property lookups yields a hit in the direct object, or its immediate prototype (Figure 11). GMail has the largest amount of non-direct and non-immediate prototype hits, but even for GMail these only add up to 10%.

We also evaluated the overall speedup due to the property cache and found that it speeds up execution of the SunSpider Benchmark set by up to 220% for certain benchmark programs (Figure 12). The overall speedup is less dramatic for some of the benchmark programs because they perform relatively fewer expensive name lookups. For very short programs (some of the SunSpider benchmark programs only run for a few milliseconds) the overhead of filling the property cache can even cause a slight performance loss.

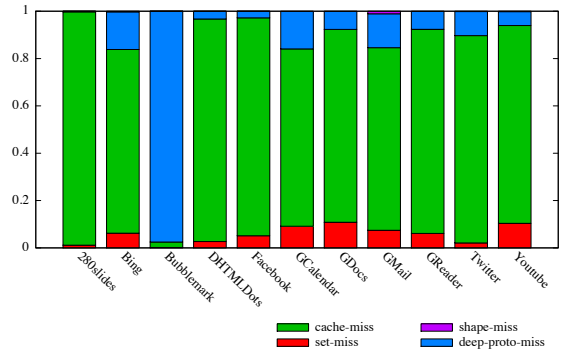

Figure 10. Distribution of cache miss causes for a set of popular web applications. The majority of cache misses are due to collisions or not yet filled cache entries. The second most frequent reason for a cache miss is *foreshadowing* of a deep prototype chain hit. Cache behavior is fairly consistent across applications. A noteworthy exception is the `Bubblemark` benchmark that uses a large array of objects with variant shapes, causing constant cache misses (and fills).

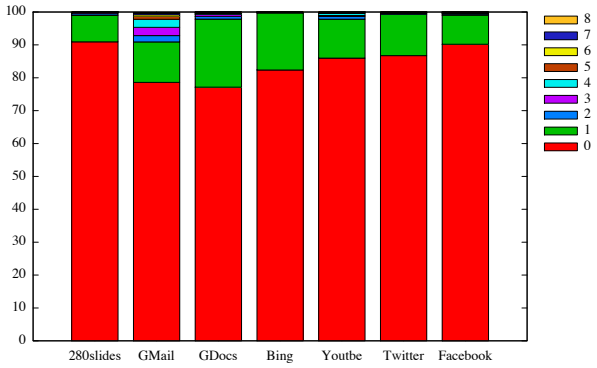

Figure 11. Distribution of property locations. The overwhelming number of properties is found on the direct object, followed by 10-20% of properties found in the immediate prototype of the object. Deep hits along the prototype chain a rare.

7. Conclusions

In this paper, *property trees* and *property caches* are introduced as a mechanism to increase the efficiency of property lookup in dynamic object-oriented languages. Property trees allow decreased memory usage through increased data-structure sharing and faster property update operations, while the property cache allows fast, cached lookup without requiring code patching. These two components are tied together by associating *shapes* with nodes in the property tree that can be efficiently guarded upon by the property cache. This paper describes details of an implementation in a production JavaScript engine. Experimental results show high cache hit rates for both computationally-expensive benchmarks and real-world web applications, as well as an overall speedup of up to 220% for certain benchmark programs.

References

[1] SpiderMonkey (JavaScript-C) Engine - http://www.mozilla.org/js/spidermonkey/.

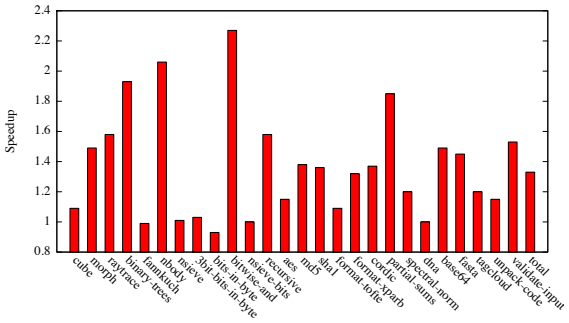

Figure 12. Speedup for the SunSpider Benchmark set. The property cache speeds up the overall execution time of JavaScript code from the SunSpider Benchmark by up to 220%. If we were to consider the time taken for the actual name lookup operations, the speedup would be even higher.

	Tests	Hits	Ratio
cube	100173	95368	95.20%
morph	435977	435961	100.00%
raytrace	637590	622400	97.62%
binary-trees	377851	377823	99.99%
fannkuch	245	239	97.55%
nbody	1473133	1472975	99.99%
nsieve	7	4	57.14%
3bit-bits-in-byte	2	0	0.00%
bits-in-byte	2	0	0.00%
bitwise-and	1200001	1199998	100.00%
nsieve-bits	3	0	0.00%
recursive	245489	245477	100.00%
aes	88265	88077	99.79%
md5	175434	174847	99.67%
sha1	162393	162361	99.98%
format-tofte	24915	14453	58.01%
format-xparb	199875	171701	85.90%
cordic	400017	399994	99.99%
partial-sums	1047612	984068	93.93%
spectral-norm	122652	122636	99.99%
dna	81	62	76.54%
base64	213012	212989	99.99%
fasta	501441	500466	99.81%
tagcloud	196486	173925	88.52%
unpack-code	94224	80318	85.24%
validate-input	294008	293960	99.98%
total	7990888	7830102	97.99%

Figure 13. Cache tests and hits for the SunSpider Benchmark set.

[2] The Perl Programming Language. http://www.perl.org/.

[3] v8 - Google Code. http://code.google.com/apis/v8/design.html.

[4] C. Anderson, S. Drossopoulou, and P. Giannini. Towards Type Inference for JavaScript. In *19th European Conference on Object-Oriented Programming (ECOOP 2005), Glasgow, Scotland*, pages 428–452, June 2005.

[5] P. André and J.-C. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 110–126, New York, NY, USA, 1992. ACM.

[6] A. H. Borning. Classes versus Prototypes in Object-Oriented Languages. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 36–40, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[7] C. Chambers. Object-Oriented Multi-Methods in Cecil. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 33–56, London, UK, 1992. Springer-Verlag.

[8] C. Chambers. Predicate Classes. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 268–296, London, UK, 1993. Springer-Verlag.

[9] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language based on Prototypes. *SIGPLAN Not.*, 24(10):49–70, 1989.

[10] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for JavaScript. *SIGPLAN Not.*, 44(6):50–62, 2009.

[11] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM.

[12] C. Dony, J. Malenfant, and P. Cointe. Prototype-Based Languages: From a new Taxonomy to Constructive Proposals and their Validation. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 201–217, New York, NY, USA, 1992. ACM.

[13] R. K. Dybvig. *The Scheme Programming Language*. MIT Press, third edition, 2002.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[15] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.

[16] D. Jang and K.-M. Choe. Points-to Analysis for JavaScript. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1930–1937, New York, NY, USA, 2009. ACM.

[17] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. *SIGPLAN Not.*, 21(11):214–223, 1986.

[18] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 235–250, New York, NY, USA, 1996. ACM.

[19] P. Steyaert and W. D. Meuter. A Marriage of Class- and Object-Based Inheritance Without Unwanted Children. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 127–144, London, UK, 1995. Springer-Verlag.

[20] D. Ungar and R. B. Smith. Self: The Power of Simplicity. *SIGPLAN Not.*, 22(12):227–242, 1987.

[21] J. Vitek and R. N. Horspool. Taming Message Passing: Efficient Method Look-Up for Dynamically Typed Languages. In *ECOOP '94: Proceedings of the 8th European Conference on*

Object-Oriented Programming, pages 432–449, London, UK, 1994. Springer-Verlag.

[22] `http://wiki.ecmascript.org`.

[23] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM.