

# Toward immediate feedback in research on virtual machines

Erick Lavoie, Marc Feeley, Bruno Dufour

Université de Montréal

{lavoeric, feeley, dufour}@iro.umontreal.ca

## Abstract

This is the text of the abstract.

**Categories and Subject Descriptors** D.3.4 [Programming Languages]: Processors—compilers, optimization, code generation, run-time environments

**General Terms** Algorithms, Performance, Design, Languages

**Keywords** JavaScript, virtual machine, compiler, self-hosted, optimization, implementation, bootstrap

## 1. Introduction

The time and efforts required from researchers to design and evaluate implementation strategies for programming languages are determining factors in the quantity and the quality of the results obtained by the whole community. Results that facilitate new virtual machines (VM) construction or modification of existing ones amplify every researcher efforts involved with programming language's implementations.

The feedback loop between the birth of an idea and its empirical evaluation is dominated by the implementation effort required to test it. When building a new VM, most of the effort is spent reimplementing functionalities with known techniques. When modifying an existing VM most of the effort is spent overcoming early-bound assumptions in the design. The implementation effort delays the feedback that can be obtained on an idea.

In our view, a research VM should provide immediate feedback on the impact of design decisions on its properties, such as performance, security and extensibility. It should do so with minimal efforts on the part of the researcher. We believe that it is a technical problem that can be solved through careful design.

We posit that the combination of the following properties can facilitate exploration and significantly contribute to the immediacy of feedback on design decisions:

- *Openness*. Any part of the system should be modifiable by user-defined code with as little constraints as possible on what can be expressed.
- *Extensibility*. New user-defined object representations should be supported with no modifications to the core of the system. New compilers should be supported by reusing parts of existing compilers or by completely replacing them without modifications to the object model.

- *Dynamism*. User modifications should be supported at runtime.
- *Performance*. The system should be fast enough to provide instant feedback on user modifications.

This paper details our first attempt at applying those principles in building a research virtual machine for an existing language. We chose to use JavaScript [4] as a source language, first because of the dire need for new implementation techniques for this particular language and second, to guarantee the applicability of our results to mainstream languages. This is our second attempt at bootstrapping a JS VM and we contrast our current approach to our previous one [1].

We first describe our current design and implementation, *Photon*. We discuss our bootstrap strategy. We perform an empirical evaluation by performing different tasks and doing a quantitative analysis both of the complexity and performance of the system. We finally compare with the current literature and identify areas for future improvement.

### 1.1 Contribution

The main contribution of this paper is in the design, implementation and evaluation of a working self-hosted JS VM exhibiting a higher level of *Openness*, *Extensibility* and *Dynamism* than other JS VMs we know of. A complementary contribution is on the choice and experience report on our bootstrap process. A final contribution is in the empirical evaluation in using a domain-specific language, OMeta [10] to write a JS to native code compiler.

Although we do not claim that Photon fully achieves the four aforementioned properties at the time of writing this article, we think that it achieves enough to clearly show the benefits of aiming for those properties in the first place.

## 2. Design

Two insights lead to understanding how the current design achieves the first three properties. Those two insights were key behind the invention of both LISP [7] and SmallTalk [5]. They are also fundamental to recent work on open, extensible object models [9] from which our design draws heavily.

The first one is *late-binding*. Late-binding is the act of deferring as long as possible the choice of a concrete strategy to implement an expected behavior. The latest time at which a given concrete strategy might be chosen is just before performing the expected behavior. By opposition, *early-binding* means choosing a concrete strategy earlier, usually at compile time. *Late-binding* is what enables runtime modifications to the system.

The second one is *uniformity*. By organizing the system around a single principle, as in "Everything is a list" or "Everything is an object", one can modify every part of the system in the same way. It reduces the possibilities of interference between different elements of the design.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPLASH 2012 October 19-26, 2012, Tucson, Arizona, USA  
Copyright © 2012 ACM [to be supplied]. . . \$10.00

*Late-binding* and *uniformity* have traditionally been associated with poor performance. However, work on SELF [3] has shown that performance could be brought within a factor of 3 from optimized C code on some benchmarks, with the right optimizations. As the JS object model is inspired from SELF, we expect those techniques to apply to JS as well. Usage of *late-binding* and *uniformity* in the design should not compromise immediate feedback because of performance issues.

Before we delve into the current design of Photon, we first briefly introduce JS, we discuss why Photon is meta-circular and justify our unified environment execution approach for user and compiler code. The design explanation focuses on the object model since it is the enabler of the first three properties in the system.

## 2.1 JavaScript

JavaScript is a dynamic language, imperative but with a strong functional component, and a prototype-based object system similar to that of SELF.

A JS object contains a set of properties (a.k.a. fields in other OO languages), and a link to a parent object, known as the object's prototype. Properties are accessed with the notation `obj.prop`, or equivalently `obj["prop"]`. This allows objects to be treated as dictionaries whose keys are strings, or as one dimensional arrays (a numeric index is automatically converted to a string). When fetching a property that is not contained in the object, the property is searched in the object's prototype recursively. When storing a property that is not found in the object, the property is added to the object, even if it exists in the objects prototype chain. Properties can also be removed from an object using the delete operator. JS treats global variables, including the top-level functions, as properties of the global object, which is a normal JS object.

Anonymous functions and nested functions are supported by JS. Function objects are closures which capture the variables of the enclosing functions. Common higher-order functions are predefined. For example, the `map`, `forEach` and `filter` functions allow processing the elements of an array of data using closures, similarly to other functional languages. All functions accept any number of actual parameters. The actual parameters are passed in an array, which is accessible as the `arguments` local variable. The formal parameters in the function declaration are nothing more than aliases to the corresponding elements at the beginning of the array. Formal parameters with no corresponding element in the array are bound to a specific undefined value.

JS also has reflective capabilities (enumerating the properties of an object, testing the existence of a property, etc) and dynamic code execution (`eval` of a string of JS code). The next version of the standard is expected to add proper tail calls, rest parameters, block-scoped variables, modules, and many other features.

## 2.2 Meta-Circularity

Using the language to compile for implementing a VM provides numerous advantages such as code reuse, runtime sharing and elimination of conflictual interactions between the client and host runtime systems. The most important advantage in our case is the possibility of intercession on the system behavior by simply exposing the inner components of the VM.

However, this raises meta-stability issues because infinite recursion can be introduced by inadvertence. While the system is in its early phases, we think it is advantageous to open the system for easy experiments because it allows a faster evolution, even at the cost of meta-stability issues. Those can be mitigated by restricting our coding style until a proper meta-object protocol [6] is designed.

## 2.3 Unified Environment

Our previous work on the Tachyon self-hosting VM for JS [1] used separate environments for the compiler and user code. Although it contributes to the security of the implementation, it complexifies both serialization of the system and intercession on the compiler. Photon uses a unified environment, where all the VM internals are exposed as JS objects on the global object, making serialization and intercession simpler.

## 2.4 Object Model

The goal of the Photon object model is to facilitate intercession on the VM behavior rather than be fully compliant to the ECMAScript standard. The deviations introduced should be invisible to programs using only common reflexion mechanisms (`typeof`, `instanceof`, `for in` iteration).

The next sections present the base JS object model and contrast it with the Photon object model. We discuss only the value types necessary for self-hosting.

### 2.4.1 JS Object Model

The JS object model has two regimes. Values in the language are either primitives or objects. Primitives comprise booleans, strings, numbers, `null` and `undefined` values. Every other value is an Object, including object versions of primitives.

The behavior of primitive values is defined by the implementation and is opaque. The behavior of object values is partially open. Prototypes for the base objects are exposed on the `prototype` attributes of their corresponding constructor function. Standard library functions are modifiable attributes of these prototypes. However, basic operations on objects such as property access and assignment are fixed. The next version of the ECMAScript standard introduces proxy objects to redefine the behavior of basic operations. They cannot the behavior of operations that do not go through them.

Some value types are special. The `arguments` object does not have a specific constructor function. It prevents addition of methods for its instances. Although its values are accessed through numerical properties and it has a `length` property, it does not support the array methods. It inherits from the root object.

The three categories of values and their prototyping relationships are illustrated at figure 1.

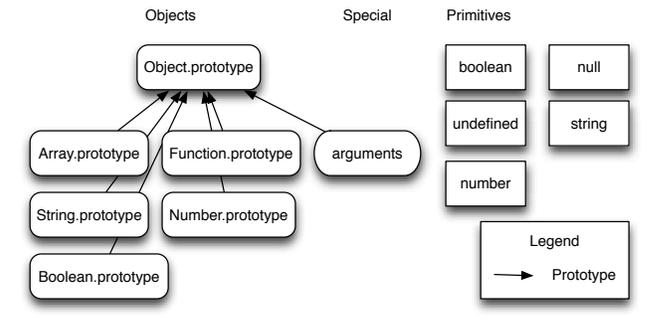


Figure 1. JS Object Model

### 2.4.2 Photon Object Model

The Photon object model is heavily inspired from the open extensible object model proposed by Piumarta and Warth [9]. The most interesting feature of that object model is its recursive definition. Components of the object model are objects whose behavior is affected by the object model. This provides encapsulation of the implementation strategy, remove external dependencies and facilitate extensibility.

Their object model defines 3 types of objects, `object`, `symbol` and `vtable`. Everything is an object. Behavior is invoked through message sending. Symbols are canonical values used to represent messages. Vtables define the behavior of objects by associating symbols with method implementations. Every object has an associated vtable. Since vtables are also objects, a vtable's vtable defines the behavior of vtables. To close the recursion, this last vtable is its own vtable.

Photon object model is similar except that method implementations are stored on objects. It unifies meta-level and JS-level methods. JS functions can redefine the object model behavior. Every component of the object model is a JS object, including the equivalent of a vtable, which we call map in the tradition of SELF's terminology. A map defines the layout of an object instead of its behavior. Maps are also objects and their behavior is defined by a prototypical map. The layout of a map is defined by its map. To close the recursion, the layout of all maps with no property is defined by a recursive map and the prototypical map is its own map. This illustrated at figure 2. The layout description of maps is immutable. Adding or removing properties from an object replaces its map. Special care must be done when adding or removing properties from recursive maps to maintain the recursive property.

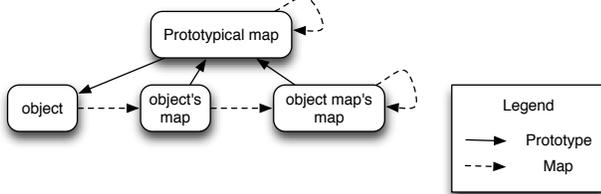


Figure 2. Photon Meta Object Model

To unify the intercession process, every value in Photon has a corresponding prototypical object, even primitive values and implementation objects. The prototypical object of `null`, `undefined`, `true` and `false` and numbers is `primitive`. The `arguments` object has a corresponding prototypical object, which is a child of the array prototypical object. All strings are symbols and inherit from `symbol`.

The last three prototypes are reifications of implementation objects. `cwrapper` is the prototype for all wrappers to c functions. `cell` is the prototype for all cell values used for implementing closures and aliasing of arguments object values to local variables. `frame` is a special object used for implementing the `apply` method for functions. The prototypical inheritance is illustrated at figure 3.

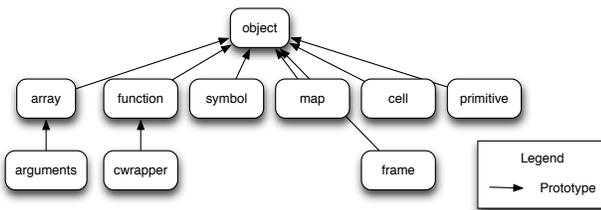


Figure 3. Photon Inheritance Hierarchy

Figure 4 illustrates the core of the object model that allows message sending. It includes an example `foo` object with a `"foo"` property whose value is the `bar` function. Prototypical objects from the inheritance diagram in figure 3 are greyed. Object values are represented in squares growing up. Associations between symbols and offsets in maps are represented under the map objects.

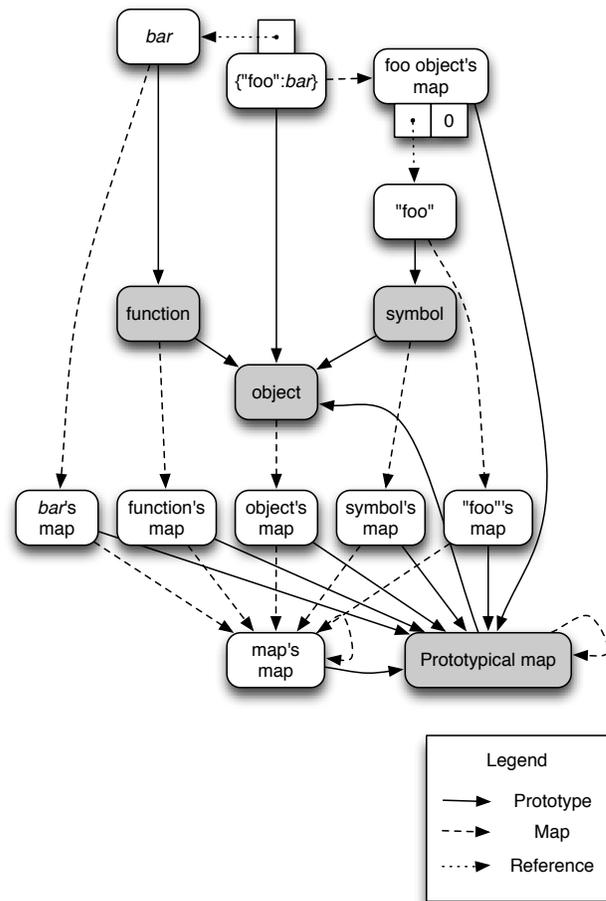


Figure 4. Photon Core Object Model

The only missing piece is the message sending algorithm. The algorithm is identical to the original suggested by Piumarta and Warth. We present it in pseudo-JS at figure 5. `===` performs an identity equality. `_objMap(rcv).lookup(rcv,msg)` sends the lookup message to the `rcv`'s map. It allows redefinition of the lookup method but needs a base case to end the recursion which is performed by the early-bound `_mapLookup` call. The following constructions are not available in ECMAScript 5 but they are used to clarify the presentation. Every variable prefixed with an underscore means that its value is bound at compile-time. `_obj*` are accessor functions to the underlying object representation. They are conceptually inlined at the call site. The `send` function is inlined at every call site. The `..args` means the rest of the arguments.<sup>1 2</sup>

This object model is *uniform*. Everything is a JS object and behavior is invoked through a message send. This object model is also *late-bound*. The semantic of message sending illustrate that the behavior invoked is bound just before its execution.

<sup>1</sup> A safe implementation should perform a check that the value found is a function before the call.

<sup>2</sup> Sending the `doesNotUnderstand` message to the current object when no method has been found instead of throwing an exception would bring the SmallTalk message sending behavior to Photon.

```

function _bind(rcv, msg) {
  if (_isPrimitive(rcv)) {
    return _primitive[msg];
  } else if (msg === "lookup" &&
    _objMap(rcv) === rcv) {
    return _mapLookup(_map, msg);
  }

  var offset = undefined;
  var rcv = rcv;

  while (rcv !== null) {
    offset = _objMap(rcv).lookup(rcv, msg);
    if (offset !== undefined) {
      return _objValue(rcv, offset);
    }
    rcv = _objProto(rcv);
  }

  if (offset === undefined)
    throw "error";
};

inline send(rcv, msg, ..args) {
  return _bind(rcv, msg)(rcv, ..args);
}

```

Figure 5. Message sending algorithm

### 2.4.3 Compiling JS to the Object Model

Compilation of JS now essentially consists in mapping JS operators to message sends. For example, `o.p` compiles to `o._get_>("p")` and `o.p = v` compiles to `o._set_>("p",v)`. Choosing unused names for JS operations keeps meta-level operations at the user level to facilitate intercession.<sup>3</sup>

### 2.5 Discussion

By virtue of being meta-circular and by exposing the compiler and prototype objects on the global object, this design is fully *open*. Any element that determine the behavior of the system is modifiable by user code.

This design is also *extensible*. New object representations can be defined using existing objects as prototypes to reuse behavior. The encapsulation provided by the recursive definition of the object model makes replacement of the existing compiler with a new one easier, as almost no assumption about the layout of objects are made. In fact, assumptions about the layout of objects can be seen as compile-time *binding* and *inlining* of object methods.

Not everything is late-bound in the current design, therefore it is not fully *dynamic*. The message sending primitive, function call protocol, type information to distinguish primitives from objects, the header layout and the position of property values in the object representation can be changed at runtime only at the cost of recompilation of all the existing code. The deeper issue here is that every early-bound element requires a recompilation of existing code when that element changes. A safe incremental recompilation technique supporting those changes could provide the illusion of dynamicity while still providing performance. To the best of our knowledge the design of such a system is still an open question.

<sup>3</sup> Should the need arise to prevent interference of user-level properties with meta-level properties, different symbol tables can be used.

However, the choice of early-binding some operations can intentionally be made. We did it in this first implementation for control flow, arithmetic, logical and relational primitives as well as mapping the environment to locations on the stack or on closures. In the context of research on programming language implementation for existing languages this seems reasonable since, as far as we know, all the mainstream languages do not allow such changes. The need might arise in the context of programming language design as one might want to experiment easily with such changes. Recent work by Piumarta [8] addresses some of these issues.

Not being entirely compliant to ECMAScript 5 might be problematic. The distribution of language features used by real world applications and websites is still an open question. Answering that question would allow us to provide the proportion of programs that could run correctly on Photon in the presence of an object model that is not fully compliant to the standard. In the absence of empirical data, we hypothesise that this proportion is high enough for Photon to be a useful tool for empirical research.

## 3. Implementation

The implementation choices were made first and foremost to minimize the effort, at the expense of memory usage and execution performance. A number of simplifications were made.

*Fixed-precision integer numbers.* The ECMAScript standard specifies that all numbers use a double-precision floating-point representation. Some implementation internally use an immediate integer representation when possible to speed up operations. We made the same choice but we do not provide a fallback to a floating-point representation.

*Inline case for fixnum and boolean operations only.* Arithmetic, logical and relational operations perform implicit conversions of value types when the expected types are not appropriate. To the exception of string concatenation (+) and string comparison with relational operators (<, <=, > and >=), implicit conversions are not supported.

*C-compatible strings, functions and calling conventions.* To facilitate reuse and integration with C code, strings can be manipulated with the `c` string library, functions are direct pointers to C-compatible code and the 32-bit C calling conventions are followed.

*Stack-based execution.* Register allocation was an important source of bugs and compilation time in our work on Tachyon [1]. It notably complexified the code generation phase and made changes to the intermediate representation harder. We decided to map every local variable to a location on the stack for the duration of the function and return the result of an expression evaluation in the EAX register. This removes the need for register allocation and makes code generation compositional, i.e. code generated for a given node does not depend on code generated for previous nodes.

*Tree-based intermediate representation.* A single intermediate representation is used to avoid conversion between representations. It also makes code generation trivial for a stack-based execution model.

*x86 32-bit support only.* Calling conventions and instruction encoding are different between the 32-bit and 64-bit versions of x86. We favored the 32-bit version because it integrated nicely with the stack-based execution model.

*Single notation for all compiler phases.* The JS version of OMeta was used to express all compiler phases. Built-in support for pattern-matching and ease of grammar extensions made prototyping easier.

*Dynamic assembler.* Serialization of dynamically generated code provides the same advantage as statically generated code. No support for static generation is made.

*Executable heap.* The whole heap is executable to unify memory management of objects and functions.



```

inline ref_is_fixnum(r)
{
    return @[["begin",
             ["get", "r"],
             ["code",
              [_op("mov", _EAX, _ECX),
               _op("and", _$(1), _ECX),
               _op("mov", _$_(TRUE), _EAX),
               _op("mov", _$_(FALSE), _ECX),
               _op("cmovz", _ECX, _EAX)]]]
    ]}@;
}

if (ref_is_fixnum(rcv))
{
    ...
}

```

Figure 8. Inline example for testing for fixnum values

used to represent a signed integer. Reference to objects always use 0 for the least significant bit. This is achieved by aligning objects to even addresses. Although it slows down arithmetic operations, it was originally done to use pointer to c functions as methods before proper support for wrappers was introduced.

The representation for objects in memory is illustrated at figure 9. Every object share the same header fields represented in gray:

- *values size*: The number of user-defined property slots available in the object
- *extension*: Reference to another object, should the number of values required exceeds the number of slots available
- *flags*: Flags used to traverse the heap
- *payload size*: The number of bytes in the object's payload
- *prototype*: The object's prototype
- *map*: The object's map

User-defined properties extend toward decreasing addresses from the header and the payload extends toward increasing addresses. The reference to an object always refers to the first byte of the payload. This scheme allows symbols and functions to be used as c-strings and c-functions as well as directly calling to functions from a function reference.

Except for the payload of strings, functions and maps, every object slot uses a boxed representation. This allows manipulation of the values with regular JS code. An interesting insight is to see objects in memory as bidirectional arrays, that can be addressed with negative indexes for access to the header or user-defined properties and positive indexes for access to the object's payload. A special syntax, [*@<index>*] is provided for that purpose as explained in section 3.3. Accessing the map of an object *o* would be written *o*[*@-1*]. Different representation examples for basic types are given at figure 10. Note that header and value fields have been compressed to a single slot to simplify the presentation.

An example method, written in C is given at figure 11. This method takes no arguments other than the defaults, which are the number of user arguments given, a reference to *this* (named *self* in the C implementation) and a reference to the closure. The *ref* function in C boxes a native signed integer. This method is used to provide reflexive information on the size in memory of the header part of the object.

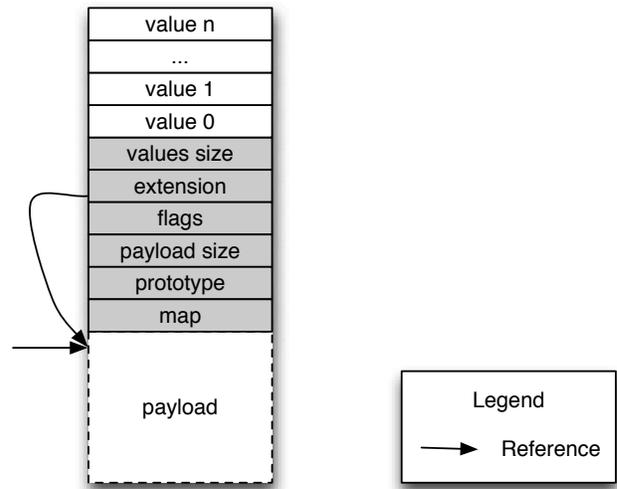


Figure 9. Object Representation

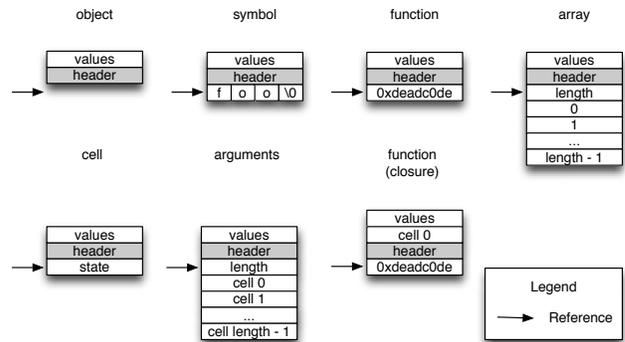


Figure 10. Object Examples

```

struct object *object_header_size(
    size_t n,
    struct object *self,
    struct function *closure)
{
    return ref(sizeof(struct header));
}

```

Figure 11. Method example

### 3.5 Compiler Implementation

The compiler uses OMeta [10] for expressing each phase of the compilation process. The rationale behind this choice was to unify the implementation of the compiler. This greatly facilitates bootstrapping because once the OMeta runtime was supported, the whole compiler was, which would not have been the case if different phases used different features of the JS language. The second advantage is that the OMeta notation is more concise and at a higher-level than JS. It opens the possibility of optimizing the compiler implementation with domain-specific invariants specific to pattern-matching and compiler implementation. Although we

have yet to exploit this opportunity, we think that being able to retarget the OMeta code to the current features and optimizations supported in the system makes the implementation easy to evolve.

The compiler phases are:

- *Parsing*: A string is converted to the array-based AST
- *Inline Expansion*: Inline calls are expanded
- *Desugaring*: JS converted to a restricted subset, in the spirit of  $\lambda_{JS}$  [2]
- *Variable Scope Analysis*: Variables are determined to be either local, captured or global
- *Variable Scope Binding*: Variables are bound either to the global object, to a local function or a closure environment
- *Optimization*: Every message send for property access or update is tagged such that the runtime can optimize it
- *Code Generation*: Native executable code is dynamically generated. A function containing the executable code is returned.

### 3.6 Discussion

Different implementation choices could be accommodated by the design presented in section 2. For example, different calling conventions, tagging schemes or object representations could be chosen while maintaining the properties of the design. More or less operations of the language could be reified as message sends to provide *late-binding* or *early-binding* depending on the desired properties of the system.

## 4. Optimizations

Most optimizations were left out during design, with the exception of *early-binding* the location of methods on objects and providing inline boolean, arithmetic and control-flow operations. Once self-hosting was achieved two optimizations were introduced. We present the first to show how the object model made it easy and explain the second to give background to the empirical results obtained. A full explanation is outside the scope of this paper.

### 4.1 Lazy creation of prototype property

Functions in JS perform many roles, notably they provide lexical scoping and serve as constructors. OMeta generated code makes an extensive use of anonymous functions to provide lexical scoping. Creating a new prototype object in addition to the function object each time an operation is performed is both space and time consuming. By redefining the property access method for functions objects we provided lazy creation of the `prototype` property easily.

### 4.2 Inline cache

An inline cache memoizes the lookup of methods by caching the result at the call site. Since regular methods and meta-methods are represented in the same way, a cache has the added benefit of optimizing both.

For a cache to be effective with an important number of objects, those objects need to be regrouped into *families*. With the current object model, two objects belonging to the same family share the same map. We chose to determine a family based on two conditions:

- *Ancestry*: Two objects from the same family have the same prototype object
- *Layout*: Two objects from the same family have the same history of property modifications (creation, deletion)

We identified two things that could be cached:

- *Method value*: The actual method object found during lookup

- *Method offset*: The offset at which the method was found

We chose to cache the method value if the value was found on an ancestor and cache the method offset if the value was found on the object itself. For simplicity, we implement a monomorphic cache.

In addition to caching the value of a method, the inline caches and the invalidation mechanism are reused to cache the location of properties on objects for retrieval and update. This is done by replacing the method value by a specialized method memoizing the offset on the object. The caches are invalidated if the layout of the object changes (because its map would change) or if the retrieval or update methods are modified. It preserves the properties of the design.

## 5. System Bootstrap

The bootstrap process is illustrated at figure 12. The first step consists in loading the Photon source code into v8. The compiler is now available for further compilation.

The second step consists in initializing a new heap for hosting the photon objects and then bringing up the object model into it. Since the object model is self-referential, a bootstrap strategy is needed to initialize it. C Methods are first used to create objects and assign them only enough functionality to use the object model operations themselves to complete the rest of the initialization. We refer the reader to [9] for further details on how to bootstrap a self-referential object model.

The third step consists in compiling Photon source code with the compiler hosted in v8. Doing so will render the Photon compiler available in the Photon heap. The bootstrapped compiler is now available for further compilation.

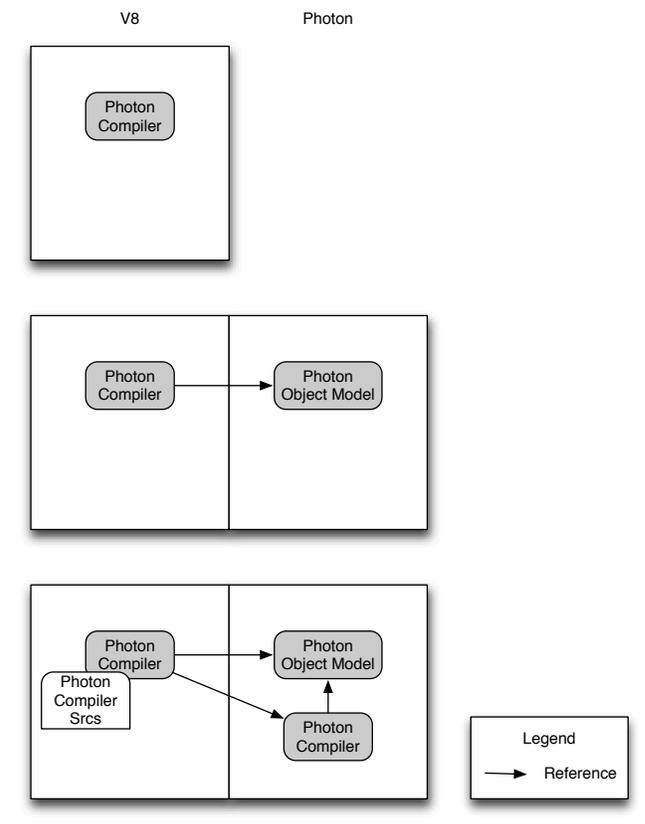


Figure 12. Bootstrap

## 5.1 Serialization

Having a meta-circular compiler implementation greatly simplifies the serialization of the whole system. Once the compiler has compiled itself, a simple serialization scheme will serialize both the compiler and the existing objects referenced by the global object. This allows the state of the compiler to be serialized with the same mechanism. In a non meta-circular implementation, a separate scheme would need to be devised for that purpose.

An interesting insight we had when thinking about the serialization process was to express the layout of objects in memory as assembly code. By guaranteeing that object do not move in memory during serialization, their current address can be used as a unique identifier. By expressing the IDs as assembly labels, the assembler utility can be used to patch addresses in the right places. An example serialization for a simple object is given at figure 13.

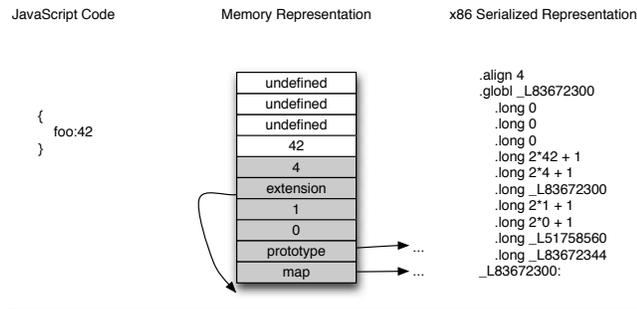


Figure 13. x86 Serialization Example

We decided to serialize the system to an x86 executable to avoid the loading phase needed when starting the system. It has the nice benefit of producing a self-contained executable in a single file. We use GCC to assemble the x86 assembly file produced during the serialization process into a native executable. Interface with the OS and C methods are kept in a C library file, which will be compiled at the same time as the x86 assembler image. The process is illustrated at figure 14. Currently, we use the exact same C file for bootstrapping with v8 and creating the executable image to avoid duplication.

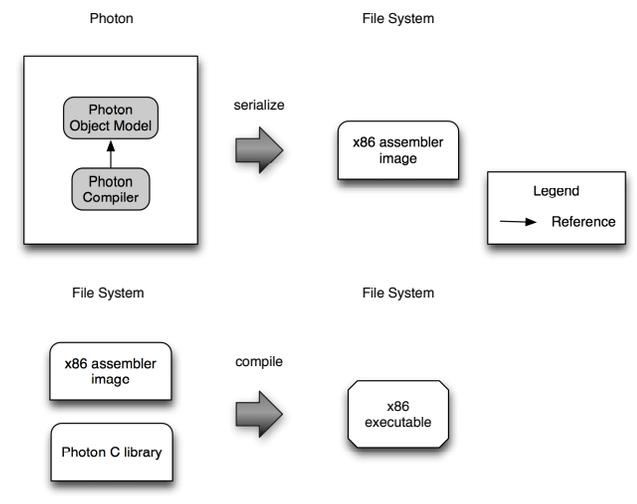


Figure 14. Serialization to x86 executable image

## 6. Use Cases

The following use cases present how different tasks are made easy. We first show *extensibility* by adding another object type. We then show *openness* by instrumenting a basic operation of the language. We show how *openness* makes tasks that do not necessarily require *dynamism* easy. We finally show how *openness* and *extensibility* make supporting other languages easy.

### 6.1 Extending the object model

The next version of the JS standard, nicknamed *Harmony*, introduces native support for associative table, which they call Map. To avoid confusion with the meta-objects in Photon, we distinguish them by using an uppercase M for the associative table name and a lowercase m for the meta-objects.

Providing native support brings a performance advantage by allowing the implementation to heavily optimize the operations of the new object type. In this use case, we show that our system is both able to profit from a direct access to memory to implement native types and reuse much of the existing system to simplify the introduction of user-defined native types.

For simplicity of presentation, we use an implementation that performs a linear search for every Map operation. We compare an implementation done in regular JS with a native implementation done using extensions presented in previous sections. The regular JS implementation is given at figure 15. It uses two arrays to store keys and values. The native implementation is given at figure 16. It uses the payload section of the object to store the number of entries in the table and the entries themselves. Doing so eliminates a level of indirection. Comparing the time required to perform 2000 times each of the `set`, `get`, `has` and `delete` operations shows a 140% slowdown between the native and the regular implementation. The running times are given at table 1.

Implementation	Photon (s)	Slowdown
Regular JS Map	2.6	1.4
Extended JS Map	1.8	1.0

Table 1. Running times (in seconds) of different implementations of the Map object

Additional considerations are required for a native implementation. First, extension of the object is necessary once the payload section is full. This is done by creating a clone of the current object with a bigger payload section and updating the extension slot of the original object. Every operation on the object then needs to retrieve the extended object before performing the actual operation. Second, the object protocol must be followed. In this case, a `__new_default__` method is necessary for correct behavior with the `new` operator. Third, implementation invariants need to be maintained. In this case, the inline cache implementation requires a common map for all objects created from the prototype. This is achieved by creating a new `map` and assigning it to the `__base_map__` property of the prototype object.

Some things are provided for free. For example, the extensible object model provides the regular object behavior without requiring any additional code for the native implementation.

Comparing the two implementations shows that direct memory manipulation and additional considerations take twice the number of lines of code for an equivalent native implementation, to provide a 30% speedup.

### 6.2 Profiling the runtime behavior

The runtime profile of a JS program is currently laborious to obtain. The common approach is to instrument an existing implementation with various hooks to record runtime events. The major problem

```

function Map() {
  var that = this;
  that._keys = [];
  that._vals = [];

  return that;
}

(function () {
  function indexOf(keys, key) {
    var length = keys.length;
    for (var i = 0; i < length; ++i)
      if (keys[i] === key) return i;
    return -1;
  }

  Map.prototype.get = function (k) {
    var i = indexOf(this._keys, k);
    if (i >= 0) return this._vals[i];
    else return undefined;
  };

  Map.prototype.has = function (k) {
    return indexOf(this._keys, k) >= 0;
  };

  Map.prototype.set = function (k, v) {
    var i = indexOf(this._keys, k);
    if (i >= 0) return this._vals[i] = v;

    var keys = this._keys;
    var l = keys.length;
    keys[l] = k;
    this._vals[l] = v;
    return v;
  };

  Map.prototype.delete = function (k) {
    var i = indexOf(this._keys, k);
    if (i < 0) return false;

    var keys = this._keys;
    var last = keys.length - 1;
    var vals = this._vals;

    if (i !== last) {
      keys[i] = keys[last];
      vals[i] = vals[last];
    }

    keys.length --;
    vals.length --;
    return true;
  };
})();

```

Figure 15. Regular JS Map implementation

with this approach is that all implementations running in browsers have been heavily optimized for performance with no concern about instrumentation.

The existence of an object protocol allows a much easier way to instrument running programs. Additionally, the ability to dynamically modify the behavior of the system makes it possible to profile the running behavior for a *part* of the execution instead of having to choose between profiling for the whole execution or no profiling at all.

The following example shows how the number of regular objects created during execution can be obtained and how the instrumentation code can be removed once it is no longer needed. The example code is given at figure 17. Running this example shows that 3006 regular objects were created for performing the evaluation of  $1 + 2$ . After reverting the method, the original behavior is restored. All the instrumentation performed is compatible with the inline cache behavior. Note that responsibility for avoiding recursion in the implementation of object-protocol methods is left on the programmers shoulders. Knowledge of the code generated by the

compiler is necessary for that purpose. In our opinion, the simplicity of the compilation model makes it reasonably straightforward.

### 6.3 Profiling compile-time information

Obtaining compile-time information is generally easier than runtime information. The common approach is to use or write a parser for the language and then reason on a defined representation. However some information, such as the number of a given operation generated by the compiler is compiler-dependent and harder to obtain since it depends on the optimizations performed.

Being able to intercede on a compiler's behavior permits the reuse of much of the infrastructure of the compiler to obtain compile-time information and it provides direct information on the actual compiler's behavior. The example given at figure 18 shows how to obtain the number and nature of message sends generated by the compiler for a given piece of code.

Again, since intercession on the compiler's behavior is done at runtime it is also possible to remove the cost of profiling when it is no longer needed without having to restart the program.

### 6.4 Changing the language being compiled

This use case explores the redefinition of a part of the compiler at runtime to support different syntaxes or semantics. The system allows completely replacing the existing compiler with a new one. For example, that could be used to provide better runtime performance for compiled code or to support a different language.

In this case, we chose to add support for a lisp dialect that uses most of the keywords and semantics of the JS language. Such an extension could allow support for syntax macros that would effectively bring syntactic extensibility to JS.

To keep reusing our beloved Fibonacci example again, an example of the syntax is given at figure 19. This is pretty much Scheme code with the exception that the `var` keyword is used to define global variables instead of `define` and the `function` keyword is used to define anonymous functions instead of `lambda`.

The implementation uses 114 lines of OMeta code to define a parser and scanner. Since the original parser is exposed on the global object, rebinding its value to the new parser effectively changes the syntax system-wide dynamically. After this change, the system can be serialized again to provide a stand-alone executable VM that understands a JS lisp-inspired dialect.

## 7. Empirical Evaluation

We perform an empirical evaluation of the resulting system to show that it is practical as an exploration vehicle for implementation techniques. We first show that our current implementation is *simple* as measured by the number of lines of code of the complete system. We then show that the implementation is *promising* in terms of performance.

### 7.1 Complexity

We use the total number of lines of code as a proxy for the complexity of the resulting system. To mitigate the effect of writing style, all the JS source code is subject to a source-to-source translation which removes all the comments and uniformize the writing style. The resulting output was chosen to approximate a hand-written coding style. For example, control statements always use curly-braces and `if` and `else` branches are written on different lines.

The system is written in OMeta, JS and C code. All the OMeta code is compiled to JS. The code required to compile the OMeta code is included in the total number of source lines of code (SLOC). The OMeta compiler is itself written in OMeta. The breakdown for SLOC is given at table 2. An interesting thing to notice is the effect of meta-circularity on the SLOC number. Since OMeta is meta-circular, the OMeta runtime used to generate the Photon Compiler

is the same as the one used by the Photon Compiler to compile JS code. Similarly, the JS runtime used by the Photon Compiler is the same as the one used by JS code to execute. That code would have been duplicated if those runtimes had not been shared. A second interesting thing to notice is the fact that almost a fourth of the SLOC are written in C. Of those, two-thirds are runtime methods for objects and the rest concern bootstrap, serialization and initialization tasks. In the future, we expect to reduce the amount of C code as more of the runtime functionalities will be expressed only as JS code.

Category	SLOC	%
OMeta Compiler(OMeta)	500	3.7
OMeta Runtime (JS)	1000	7.5
Photon Compiler (OMeta+JS)	5000	37.3
Photon Optimizer (OMeta+JS)	500	3.7
Runtime (JS)	3200	23.9
Runtime (C)	2000	14.9
Bootstrap, Serialization, etc. (C)	1000	7.5
V8 Integration (C)	200	1.5
Total	13400	100

**Table 2.** Source lines of code breakdown by component

We used OMeta code to drastically reduce the number of lines of code to write and modify. As shown in table 3, for the Photon Compiler we gain a factor of 6 in expressivity compared to having written the same OMeta code by hand. The main gain was obtained through the concise notation for pattern-matching on arrays afforded by OMeta.

Category	OMeta SLOC	JS SLOC	Expansion factor
OMeta Compiler	500	5000	10.0
Photon Compiler	1100	7200	6.5
Photon Optimizer	10	60	6.0
Total	1610	12260	7.6

**Table 3.** Expansion factor for OMeta code

Since we needed extensions to JS for writing some of the VM components, it begs the question of what proportion of the VM was written using only regular JS code. This figure is interesting because it gives a rough approximation of the amount of JS code that can be reused should the object representation or the extension syntax changes. If we also count the OMeta code, also compiled to regular JS, table 4 shows that 65% of the total code could be reused should the internal design of the VM changes. Let’s note however that assumptions *semantically* encoded in regular JS code are not apparent here but we can say from experience that these represent a small part in terms of SLOC.

Category	SLOC	%
OMeta	1600	12
Regular JS	7100	53
Extended JS	1500	11
C	3200	24
Total	13400	100

**Table 4.** Proportion source lines of code by language

Comparison to one of our previous attempts at a JavaScript meta-circular VM[1] shows a reduction of a factor of 5 in terms of lines of code (13.4KLOC vs 75KLOC). The reduction in the SLOC number can mostly be attributed to the usage of OMeta as well as the elimination of the SSA representation and register allocator. Comparison to V8 shows a factor of 28 (13.4KLOC vs

375KLOC) although to be fair, V8 provides a complete support for JS and backends for x86 32-bit and 64-bit as well as ARM. Using the SLOC number as a proxy shows that Photon is a much *simpler* system than other alternatives.

## 7.2 Performance

Performance was not the focus of the current implementation. We still compare our system to a state-of-the-art implementation to provide ballpark figures of its current performance. We show execution times for common sunspider benchmarks as well as v8-hosted and self-hosted compilation time. We then give explanations for the current bottlenecks and we propose implementation strategies to remove them.

All the numbers were obtained on an early 2011 MacBook Pro running OS X Lion 10.7.3 with a 2.2 GHz Intel Core i7 and 8 GB of memory. Both our system and d8 were compiled for x86 32-bit. We used revision 7928 of d8 and the 0.9.1 sunspider benchmarks.

Since our system does not support equality comparison (`==`), those were converted to identity comparisons (`===`). Code executing in the global environment was also put in a function to permit serialization of the compiled code when running inside Photon. It was done to avoid compilation when running tests. The same code is used for tests both on V8 and Photon. Those two modifications did not significantly affect the running time on V8.

### 7.2.1 Execution Time

We show some sunspider benchmarks results. As a reminder, we do not optimize constant arithmetic operations, every local variable is accessed on the stack and we follow the 32-bit calling conventions of C. Given these, the results look promising, since there is a lot of room for improvement using known techniques.

Two variations of some tests are introduced to illustrate the current bottlenecks of Photon. In the first one, we removed the string creation in the `access-fannkuch.js` benchmark. The resulting string was not used in the benchmarks, therefore it does not change the algorithm. We label the modified benchmark with (no string). In the second one, we replace the new operation in `access-binary-tree.js` with an internal cloning operation of a representant object with all the expected properties. We label the modified benchmark with (cloning). This benchmark cannot be run on V8 anymore, we therefore compare the running time on Photon with the original running time on V8.

Table 5 shows the relative speed of Photon and V8. We compare the speed of our system using inline caches (ic) to the speed of V8. This simple implementation strategy brings our system within a factor of 3 of V8 on a function call intensive benchmark such as `controlflow-recursive.js`. Our system is within a factor of 14 of V8 if we do not consider the two degenerated cases.

Test Name	Photon (ic)	V8	Slowdown
<code>controlflow-recursive.js</code>	1.845	0.584	3.16
<code>access-nsieve.js</code>	3.427	0.542	6.32
<code>access-fannkuch.js</code>	89.417	1.835	48.72
<code>access-fannkuch.js (no string)</code>	18.389	1.804	10.19
<code>access-binary-tree.js</code>	58.940	0.360	163.72
<code>access-binary-tree.js (cloning)</code>	5.000	-	13.88

**Table 5.** Execution time for 400 iterations of some sunspider benchmarks (seconds)

The two slowest running times can be explained as follow. During the original `fannkuch` benchmark, a linear search is performed for string internalization each time a string is created, making the running time proportional to the number of strings in the system. During the original `binary tree` benchmark, the creation of an object within Photon performs an introspective search on the prototype to

find an existing base map as a JS property to maintain the inline cache invariants. Potential solutions to these two problems will be addressed in the discussion section.

Table 6 shows the effect of inline caches on execution time. Given the pervasive use of message sends in the object model implementation to late-bind object behavior, it is of little surprise that inline caches (ic) give a huge boost to performance. When not using them (noic) the system is between 4 and 51 times slower.

Test Name	Photon (noic)	Photon (ic)	Slowdown
controlflow-recursive.js	4.606	0.195	23.62
access-nsieve.js	17.716	0.346	51.20
access-fannkuch.js	84.504	9.079	9.30
access-fannkuch.js (no string)	78.079	1.835	42.55
access-binary-tree.js	23.428	5.973	3.92
access-binary-tree.js (cloning)	15.613	0.507	30.79

**Table 6.** Execution time for 40 iterations of some sunspider benchmarks (seconds)

### 7.2.2 Compilation Time

Table 7 gives the compilation time for two of the precedent benchmarks. It shows a factor of 100 between the self-hosted and the v8-hosted version. From the result given in the previous section, we believe this is mostly caused by the speed of the object creation protocol.

Additionally, bootstrapping the current system on V8 currently takes around 30 seconds. Given the slowdown factor when performing self-hosted compilation, we did not attempt a self-hosted bootstrap.

Test Name	Photon (ic)	Photon (noic)	Photon/V8
controlflow-recursive.js	10.780	8.858	0.085
access-nsieve.js	10.022	8.103	0.078
Total			

**Table 7.** Compilation Time for some benchmarks (seconds)

Table 8 gives the relative compilation time for each compilation phase. When v8-hosted compilation is done, parsing dominates the compilation time. However, when self-hosted compilation is performed, code generation is dominating by a factor of 10 in the worst case over parsing. We believe this is also because the object creation protocol is slow. Usage of inline caches actually negatively impact performance because more objects are created during code generation.

Phase	Photon (ic)	Photon (noic)	Photon/V8
Parsing	9	11	48
Inline Expansion	0	0	2
Desugaring	0	0	2
Variable Analysis	0	0	2
Variable Scope Binding	0	0	4
Optimization	0	0	4
Code Generation	87	86	33

**Table 8.** Compilation Time Breakdown in % for controlflow-recursive.js

### 7.2.3 Discussion

Both execution time and compilation time performance are determined mostly by the limited time we had to optimize the implementation, not by limitations in the design. We provide here possible solutions to the problems identified and we show that with reasonable efforts the system could be made *practically* fast.

The previous results indicate that object creation and string internalization are major bottlenecks of the current implementation. Object creation could be sped up using a caching strategy for the initial creation of the object and subsequently for the field initialization. String internalization can be made amortised constant time instead of linear time by using a hash map. Those two problems will be addressed in the near future since the implementation effort required is minimal.

Once all the common object model operations will be appropriately cached, the next step to reduce the execution performance gap between V8 and our system will be to provide optimized instructions for constant arithmetic operations and combine test in expressions with the code generated for the conditional branching. This should also mitigate the absence of a register allocator by preventing access to the stack for binary operations with a constant.

Compilation time can be reduced by preencoding patterns of instructions for each corresponding AST node. Encoding is currently performed by making calls to the in-memory assembler for each instruction to encode. Encoding currently allocates many objects in the heap. Preencoded patterns could be exposed as functions that would patch values in a array mostly composed of integers. This will reduce self-hosted and v8-hosted compilation time since the number of created objects will be drastically reduced.

The compilation time breakdown for v8-hosted compilation shows that parsing is the dominant factor time-wise, followed closely by code generation. Further inquiry in the runtime behavior of OMeta grammars will be needed to see if this is something that could be optimized with changes to the grammar or with a more sophisticated OMeta compiler or if limitations in the design of OMeta would prevent such optimizations.

## 8. Experience Report

This experience taught us that bootstrapping a new self-hosted system is a compromise between simplicity of implementation and performance of the resulting system. Although our current implementation is too slow to serve for its own bootstrap, being able to bootstrap the system on V8 in 30 seconds allows fast experiments to be made to determine the origin of performance bottlenecks. Our previous experience with Tachyon was that 3 to 8 minutes were required for bootstrap, drastically reducing the speed at which experiments could be performed. In our opinion, this was mostly caused by the size of the code base and the really rich internal representation used to represent code, which required an important number of objects to be allocated.

This suggests that when bootstrapping a self-hosted system, the first thing to do is to keep the implementation as simple as possible to minimize the quantity of code to compile. Bootstrapping time can be minimized by borrowing performance from existing optimized implementations, in our case, the V8 implementation for compiling the system and the C compiler to provide method behavior for objects. Should an existing optimized implementation did not exist, a fast compiler could be written in C. Compilation time can then be reduced by optimizing the bootstrap compiler. In our case, it is the same as the self-hosted compiler so this optimization benefits both. Once bootstrap is near instantaneous, experiments can be performed to find and address performance bottlenecks until performance becomes a non-issue for all common development tasks.

For future work on Photon, the first thing to optimize would be the compilation speed on V8 by reducing the number of objects created, then addressing the speed bottlenecks introduced by OMeta until the bootstrap on V8 is near instantaneous. The next thing to optimize will be the self-hosted compilation speed until it is also near instantaneous for incremental modification of the live system. That will allow fast experiments to be made until self-hosted bootstrap is near instantaneous. At that time, the system will be

truly independent of existing implementations and free to evolve by itself to explore different implementation strategies. Bootstrapping speed will not be a limiting factor in the exploration of VM implementation.

## 9. Related Work

The design for an open and extensible object model has already been proposed by Ian Piumarta and Alessandro Warth [9]. However, its applicability to existing languages has not been verified empirically. The main contribution of this paper is to provide a case study for taking an existing dynamic language, JavaScript, and turn it into an open and dynamic implementation where user-defined functions can modify its behavior at runtime. We confirm the resulting flexibility by showing that various extensions and profiling tasks are made simple and we perform an empirical evaluation of the resulting system. The choice of examples and the empirical results presented aim to illustrate the suitability of the VM for research purposes.

TODO: comparison to smalltalk, factor, self, pypy, lively kernel, rubinius, maru, kernel, slate

See: MOP in JavaScript, see OOPSLA 2011

## 10. Future Work

The simplifications made during the implementation effort are not mandated by the proposed design. Empirical evaluation of other implementation options will allow a better understanding of the impact on performance of the current design. One notable item of interest will be to measure the overhead of also allowing redefinition of fixnum and boolean operations at runtime and removing the object layout assumption in the bind operation.

## Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) and Mozilla Corporation.

## A. Appendix

### A.1 JavaScript reserved properties

big table with all the reserved properties

## References

- [1] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour. Bootstrapping a self-hosted research virtual machine for javascript: an experience report. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 61–72, New York, NY, USA, 2011. ACM.
- [2] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. *ECOOP 2010–Object-Oriented Programming*, pages 126–150, 2010.
- [3] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 18:355–400, July 1996.
- [4] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, Dec. 1999.
- [5] A. C. Kay. The early history of smalltalk. In *The second ACM SIGPLAN conference on History of programming languages*, HOPL-II, pages 69–95, New York, NY, USA, 1993. ACM.
- [6] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.
- [7] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, Apr. 1960.
- [8] I. Piumarta. Open, extensible composition models. In *Proceedings of the 1st International Workshop on Free Composition*, FREECO '11, pages 2:1–2:5, New York, NY, USA, 2011. ACM.
- [9] I. Piumarta and A. Warth. Self-sustaining systems. chapter Open, Extensible Object Models, pages 1–30. Springer-Verlag, Berlin, Heidelberg, 2008.
- [10] A. Warth and I. Piumarta. Ometa: an object-oriented language for pattern matching. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 11–19, New York, NY, USA, 2007. ACM.

```

function Map() {
  return this;
}

(function () {
  function indexOf(m, key) {
    var length = m[@length_offset]*2;
    for (var i = first_entry_offset; i < length; i += 2)
      if (m[@i] === key) return i;
    return -1;
  }
  function capacity(m) {
    return (m[@-3]/sizeof_ref - first_entry_offset)/entry_size;
  }
  function payload_size(capacity) {
    return (capacity*entry_size + first_entry_offset)*sizeof_ref;
  }
  function extend(m, cap) {
    var that = m[@-5];
    var copy = that._clone_(payload_size(cap));
    var length = that[@-3] / sizeof_ref;

    for (var i = 0; i < length; ++i)
      copy[@i] = that[@i];

    m[@-5] = copy;
    return copy;
  }

  var length_offset = 0;
  var first_entry_offset = length_offset + 1;
  var init_nb = 10;
  var entry_size = 2;
  var sizeof_ref = this._ref_size_();
  var init_payload = payload_size(init_nb);

  Map.prototype = Map.prototype._clone_(
    (first_entry_offset + entry_size)*sizeof_ref
  );
  Map.prototype[@0] = 0;

  Map.prototype._new_ = function () {
    var that = this._init_(0, init_payload);
    that[@-1] = this._base_map_;
    that[@-2] = this;
    that[@length_offset] = 0;
    return that;
  }

  Map.prototype._new_default_ = Map.prototype._new_;
  Map.prototype._base_map_ = Map.prototype[@-1]._new_();

  Map.prototype.get = function (k) {
    var that = this[@-5];
    var i = indexOf(that, k);
    if (i >= 0) return that[@i+1];
    else return undefined;
  };

  Map.prototype.has = function (k) {
    return indexOf(this[@-5], k) >= 0;
  };

  Map.prototype.set = function (k, v) {
    var that = this[@-5];
    var i = indexOf(that, k);

    if (i >= 0) return that[@i+1] = v;

    var length = that[@length_offset];
    var cap = capacity(that);

    if (length === cap) that = extend(this, 2*cap);

    var i = 2*length + first_entry_offset;
    that[@i] = k;
    that[@i + 1] = v;
    that[@length_offset]++;
    return v;
  };

  Map.prototype.delete = function (k) {
    var that = this[@-5];
    var i = indexOf(that, k);
    if (i < 0) return false;

    var length = that[@length_offset];
    var last = 2*(length-1)+first_entry_offset;

    if (i !== last) {
      that[@i] = that[@last];
      that[@i+1] = that[@last + 1];
    }

    that[@length_offset]--;
    return true;
  };
})();

```

Figure 16. Extended JS Map implementation

```

var count, reset;

function instr(f) {
  var counter = 0;

  var g = function () {
    counter++;
    return f.call(this);
  };

  revert = function () {
    return f;
  };

  count = function () {
    return counter;
  };

  return g;
}

Object.prototype._new_ = instr(
  Object.prototype._new_
);
eval("1+2");
print("Object.prototype._new_ called "
  + count() + " times");
Object.prototype._new_ = revert();
o = {};

```

Figure 17. Dynamic profile of the number of regular objects created

```

var revert;

function instr_gen_send(f)
{
  var g = function (nb, rcv, msg, args, bind_helper)
  {
    print("Generating_send_" + msg + "");
    return f.call(this, nb, rcv, msg, args, bind_helper);
  }

  revert = function ()
  {
    return f;
  };

  return g;
}

PhotonCompiler.context.gen_send = instr_gen_send(
  PhotonCompiler.context.gen_send
);
eval("function_fib(n)_{" +
  "if_(n<2)_return_n;" +
  "return_fib(n-1)+fib(n-2);" +
  "}");
print(fib(10));
PhotonCompiler.context.gen_send = revert();

```

Figure 18. Static profile of message sends generated

```

(var fib (function (n)
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2))))))
(print (fib 40))

```

Figure 19. Lisp-inspired syntax example for JS