Performance from Aligning Smalltalk & Javascript Classes

Dave Mason
Ryerson University
350 Victoria Street
Toronto, ON, Canada M5B 2K7
dmason@ryerson.ca

ABSTRACT

Amber is a wonderful Smalltalk programming environment that runs on top of Javascript, including a browser-based IDE and compiler, as well as command-line support. The only challenge is that execution performance can be 1–2 orders of magnitude slower than native Javascript code. AmberDirect is a series of modest changes to the compiler and some infrastructure classes and methods that bring most generated programs to within a factor of two of native Javascript.

The challenge we faced was maintaining a seamless integration into existing Javascript classes while maximizing fidelity to Smalltalk execution semantics.

1. INTRODUCTION

Amber[9] Smalltalk is a programming environment created to support programming on the web (or on servers using NodeJS). It provides a Smalltalk IDE, and supports programming in Smalltalk, while automatically compiling the Smalltalk code to Javascript to utilize the native Javascript engine provided by most modern browsers.

In development mode, Amber creates a context within each function to facilitate traditional Smalltalk error reporting. This context-creation can be turned off for production mode to minimize the overhead.

Amber is also specifically designed to interoperate with normal Javascript as seamlessly as possible.

Despite these goals and optimizations, Amber unfortunately suffers from an order-of-magnitude (or more) slowdown relative to straightforward Javascript code, which can limit its usability for some applications.

We took a step back to look at how much performance could be recaptured and if the interoperation with Javascript could be enhanced

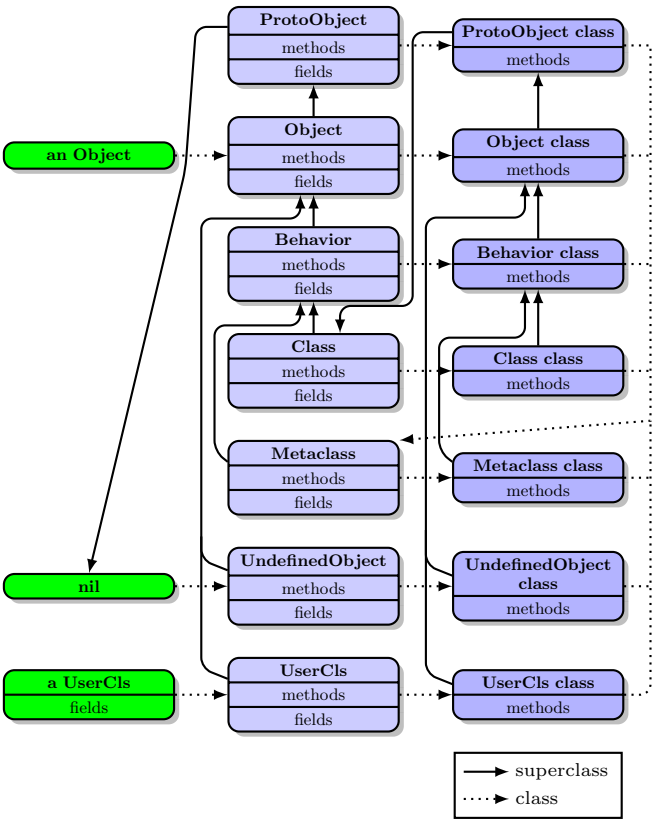

Figure 1: Classic Smalltalk Class Structure

2. EXECUTION MODEL

2.1 Smalltalk

Since the 1980s, Smalltalk[4] has had the basic structure of required classes as shown in figure 1. Ultimately, every standard object inherits from `Object`. In fact, most environments include a `ProtoObject` class that resides above `Object` and handles objects that are outside the standard environment, such as objects that have been swapped out of memory.

Smalltalk has a class-based inheritance model. Every object is an instance of a class – even classes. The methods for each object are stored in a method table associated with the class. Since even classes are instances of a class, their

methods are also in the method table associated with their *meta*-class.

Metaclasses are themselves instances of a class called `Metaclass` which ultimately inherits from `Object`, like every other class.

2.2 Javascript

Unlike Smalltalk, Javascript has a prototype-based inheritance model, essentially a simplified version of the model used by `Self`[12].

Objects are traditionally created with the `new` operator applied to a function. The result is a new object that has a field called `__proto__` that is a link to the `prototype` field of the function and a field called `constructor` that is a link to the function. If a referenced field is not in the object, the `__proto__` chain is followed to find it. See figure 2.

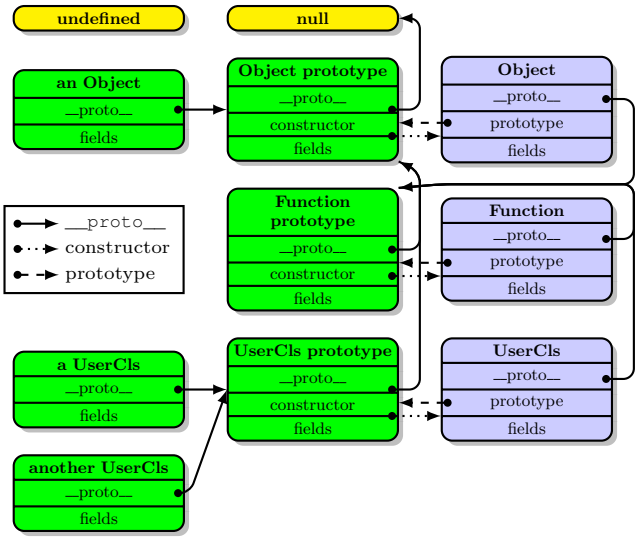

Figure 2: Javascript Prototype Structure

Here is an example from a javascript tutorial[7].

```
// Animal
function Animal(name) {
  this.name = name
}
Animal.prototype = {  // methods
  canWalk: true,
  sit: function() {
    this.canWalk = false
    alert(this.name + ' sits down.')
  }
}
// Rabbit
function Rabbit(name) {
  this.name = name
}
Rabbit.prototype = inherit(Animal.prototype)
Rabbit.prototype.jump = function() {   // methods
  this.canWalk = true
  alert(this.name + ' jumps!')
}
// Usage
var rabbit = new Rabbit('Sniffer')
rabbit.sit()   // Sniffer sits.
rabbit.jump()  // Sniffer jumps!
```

As you can see, the `prototype` object acts in a similar role to the Smalltalk method table.

The `new` operator can only be applied to functions, so it is difficult to map the classical Javascript function/prototype-/object model to the Smalltalk class model – despite being tantalizingly close – in particular, there are cycles in the Smalltalk class graph that cannot be replicated.

2.3 Amber

Amber appears to have as a goal compatability with the largest possible set of browsers. This led them to particular choices:

- instances of `Number` are represented by the Javascript *number* objects;
- `Boolean`, and `Date` are similarly directly mapped;
- `String` and `Character` both use Javascript *strings*;
- `OrderedCollection` is mapped to Javascript *arrays*;
- to avoid name clashes, and map to valid Javascript identifiers, message-names are prepended with _ and have every colon (`:`) replaced by _ ;
- to avoid name clashes, and because Javascript identifiers have only a single look-up mechanism, instance variables are prepended with `@`, which means they need to be looked up via the indexing method (`obj['@foo']`) because `obj.@foo` is invalid syntax.

There are several properties of Smalltalk semantics that are challenging to emulate in Javascript.

1. Everything is an object. This is almost true in Javascript, too, but there are 2 values: `null` and `undefined` that have no fields. They serve roles equivalent to Smalltalk `nil` but have to be converted to the `nil` object in order to send messages.
2. Messages sent to an object for which the object does not have a method send a special `doesNotUnderstand:` message to the object, which the object can implement and return, for example, a default value. In Javascript, a reference to a field that is not defined returns a value of `undefined` which raises an exception if it is applied as a function.
3. Boolean only includes `true` and `false`; otherwise signal a `mustBeBoolean` error. In Javascript there are several additional values that are treated as `false` (`undefined`, `null`, `NaN`, `0`, and `""`). *Everything* else evaluates to `true`.
4. `self` refers to the current object and `super` refers to the current object, but with method resolution starting with the superclass of the current code. In Javascript, `this` refers to the object from-which the name lookup was done that lead to the current function executing.
5. A return from a Smalltalk block returns from the method in which the block is statically defined.

Amber has solutions to all of these problems, but in most cases they are suboptimal – in part because of the choice to support legacy Javascript implementations. In section 2.4 we will look at better solutions.

Amber has a significant amount of code implemented in Javascript, both in the `support/` environment and also in the Kernel classes. This appears to be largely because of decisions about the structure of the class hierarchy and the handling of interfaces with the world of Javascript classes. Before the ECMAScript-6 (ES6) draft[2] was created, the prototype chain for Javascript objects (now called `__proto__`) was not officially programmatically available, and Internet Explorer versions before 11 do not fully support it. This means that it was impossible to interject, say, a class between `Number` and `Object` to represent Smalltalk "`Object`". This pretty-much dictates Amber's choice to use mixin methods and to use `JSProxyObject` to map non-smalltalk objects, as well as to handle `doesNotUnderstand`.

2.4 Amber-Direct

For Amber-Direct, we decided that we would abandon legacy Javascript/ECMAScript engines and try to get performance close to that of native Javascript. There are several decisions that bear upon this.

Class Hierarchy

With the arrival of ES6, the prototype chain becomes open to manipulation. We made the following decisions.

- Emulating the Smalltalk hierarchy with very good fidelity was more important than legacy browser support. Close emulation would allow more fine-grained control over where code ended up.
- The Smalltalk ProtoObject class will be the Javascript Object "class". Every non-Amber object will inherit from that class, so any methods inserted into ProtoObject will be available to Javascript objects as well as Smalltalk objects.
- Methods like `isNil` are put in ProtoObject so that interoperation between Javascript and Smalltalk objects will be more seamless.
- Rather than create the `JSProxyObject` class and wrap Javascript objects in that class, we directly interface with the Javascript objects.
- The support for direct handling of Javascript object value setters and getters and method calls is also in ProtoObject.

Object, Class, Method table relationship

As shown in figure 3, our implementation of Smalltalk in Javascript is as follows:

1. The Smalltalk object is represented by a Javascript object. The instance variables are fields in the object.
2. The `__proto__` for the object points to an object that is the method table for the class. The `__proto__` field of all instances of a class point to the same object.
3. The `__proto__` field for the method table points to the method table for the superclass.
4. The method table contains a field, `constructor`, that points to the class object.
5. The class object contains a field, `prototype`, that points to the method table for the class.
6. The class of an object, `x`, is accessed as `x.constructor`.
7. The superclass of a class object, `X`, is accessed via the reference `X.prototype.__proto__.constructor`. The only exception is if `X.prototype.__proto__` is null, in which case the superclass is `nil`.
8. A class object also contains a field, `constructor` that points to the `Metaclass` class object. Ordinary objects don't have this value, because their constructor reference is in the `__proto__`.
9. Class methods, that would be in the metaclass method table, are directly in the class object because there is only one instance of each class object, so there would be no advantage to creating a separate method table.

3. METHODOLOGY

We looked at key operations in the Javascript produced by Amber and created micro-benchmarks to explore alternate code generation choices.

For each micro-benchmark, we have a loop of at least 100,000 iterations for each of the code options. This is referred to as a run. The plots (e.g. figure 5) reflect 10 runs of each micro-benchmark. The diamonds represent the average of the 10 runs. The box runs from the 1^{st} quartile to the 3^{rd} quartile and the whiskers show extremal points withing 1.5 times that range of the median. Outlier runs show as $\times$ symbols. For most engines and micro-benchmarks, the runs are very repeatable, with the occasional outlier probably representing a major garbage collection.

We are assuming the ability to manipulate the `__proto__` chain, either by assignment to `__proto__` or by the function `setPrototypeOf()`. `setPrototypeOf()` is part of ES6, but some engines have supported assignment to `__proto__` for some time. Because of this, the only version of Internet Explorer suitable was IE11. IE11 is the default browser on Windows 8.1, but initial experiments exhibited extremely poor performance. Then we noticed that performance improved with repeated execution of the same code (from 2x to 20x speedup), so for each micro-benchmark we execute 10 "warmup" runs before we start calculating statistics – on all engines. Table 1 shows the minimum version of the tested browsers that can run our generated code and the date that version became available. Note that our definition of "Modern" browsers actually refers to browsers that have been available for over a year.

These experiments were not intended to be comparisons of browser performance, but rather to determine which of a variety of possible translations of a particular set of Smalltalk semantics runs the fastest on a particular browser. Therefore the values are scaled so that 100 is the average of the medians of the experiments for each code choice, for each

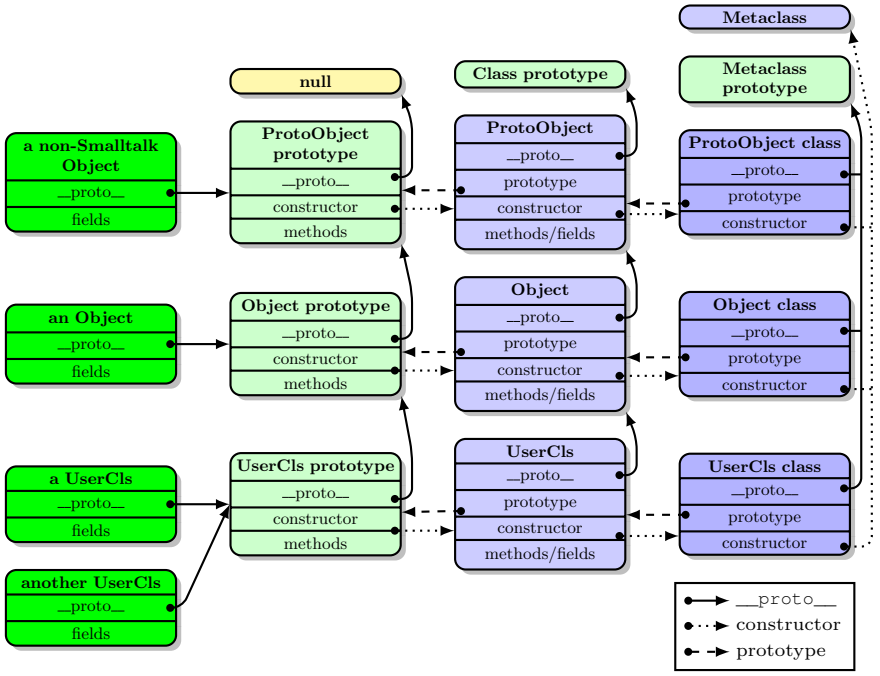

Figure 3: Amber-Direct Smalltalk Class Structure

Browser	Test Version	Min. Version[5]	Release Date
Chrome	43.0.2357.65	34	2014-04
Firefox	38.0.1	31	2014-07
Internet-Explorer	11.0.19	11	2013-10
NodeJS	0.12.2	0.10?	2013
Opera	29.0	13	2013
Safari	8.0.6	7.0?	2013

Table 1: Browser/Engine Versions

browser. Hence a low whisker diagram doesn't mean that that engine was faster, merely that that code choice was faster than the other code choices *for that engine*.

All the experiments were run on an otherwise idle Apple Macbook Pro, with 2.5GHz Intel i5 processor. IE11 was run via a virtual machine. Because we are measuring relative performance for various code choices, rather than raw performance, any residual variability should not bear on the results.

4. RESULTS

In this section we discuss each of the four areas where we have made significant changes to the Amber compiler and/or runtime environment.

4.1 Booleans

Whenever a boolean test is being done non-booleans need to be treated as errors. One might expect the `instanceof` operator to be the fastest – and figure 4 shows it is (or close) for Firefox, IE11 and Safari, but it is the slowest by far for Chrome and NodeJS. Using the `constructor` field is the fastest for IE11, but the worst by far for Safari. `amber` is the solution used by Amber, which is a function that performs the `typeof` test. Inlining `typeof` is the best performance (or very close) for all the engines, so is the one we choose to use.

Additionally, where the value is being used in an `ifTrue:IfFalse:` the code can be converted to:

```
1 if (v===true) {...true block...}
2 else if(v===false) {...false block...}
3 else v.nonbool();
```

4.2 `null`, `undefined` and `doesNotUnderstand:`

Amber handles the `nil`-equivalent values and native Javascript objects by calling a function called `$recv` on every object before calling a function on that object.

```
1 function $recv(o){
2     if (o == null) return nil;
3     if (typeof o === "object" ||
4         typeof o === "function") {
5           return o.klass != null ? o :
6               globals.JSObjectProxy._on_(o);
7     }
8     return o;
9 }
```

This function does two things.

1. Wrapping every non-Smalltalk object in a `JSProxyObject`. The `JSProxyObject` class has implementations for all possible method names that it then trampolines to access the fields or functions in the object. We have moved this to `ProxyObject`, so that any Javascript object gets the support directly. When one of the un-

amber	asBool(v)
constructor	(v.constructor===Boolean?v:v.nonbool())
eq	(v===true||(v===false?v:v.nonbool()))
instance	(v instanceof Boolean?v:v.nonbool())
typeOf	(typeof v==="boolean"?v:v.nonbool())
valueOf	Boolean.prototype.valueOf.call(v)

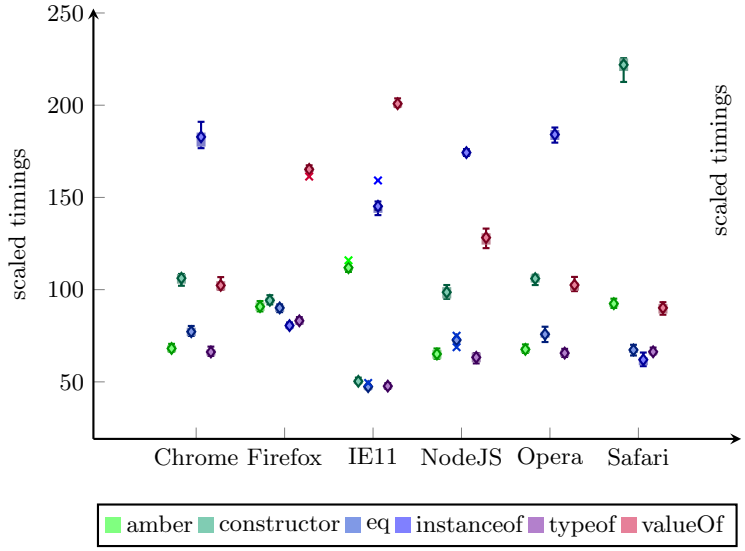

Figure 4: Recognizing Boolean values

dotget	o.ghi
dotput	o.ghi=17
idxget	o['ghi']
idxput	o['ghi']=17

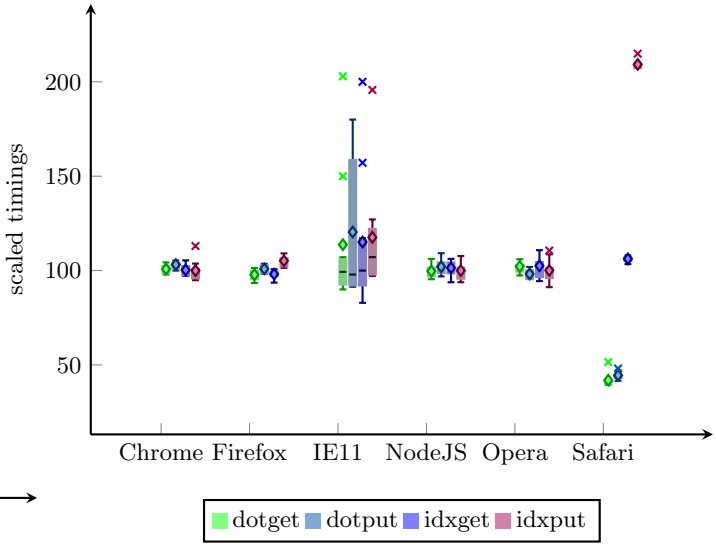

Figure 7: Access to instance variables

known functions is called, it looks through the prototype chain for the object to find where the field or function is located, then adds a new function into the same (prototype) object so that the next call will find it directly. For example, if there was an object constructor like:

```
1 function MyObject() {
2    this.abc=4;
3 }
4 MyObject.prototype.getter=function getter() {...};
5 MyObject.prototype.setter=function setter(x) {...};
```

Then a call `obj abc` (where `obj` is the result of a Javascript `obj=new MyObject()` would translate into `obj._abc()` which would dispatch to the `_abc` method in ProtoObject, which would insert into `obj` (because that's where `abc` was found) the function `function(){return this['abc']}` with the name `_abc`, and then call the function. The next time a call is made, it will find the method, `_abc`, directly in `obj` with no overhead. One of the benchmark calls looks like:

```
1 getvar: obj for: n
2    | tot |
3    tot := 0.
4    n timesRepeat: [
5       tot := tot + obj abc
6    ].
7    ^ tot
```

On the other hand, when a call like `obj._getter()` executes, it would again dispatch to the `_getter` method in ProtoObject, but in this case it would insert `_getter` in MyObject.prototype as an alias to `getter` so that subsequent references like `other getter` to that method of other instances of ProtoObject would directly execute that code without any intervention from the `_getter` method in ProtoObject.

As can be seen from figure 5, the direct calls have essentially no cost, whereas the calls via `JSProxyObject` have significant overhead.

2. Returns `nil` where necessary. Because the first reason has been eliminated, Table 6 shows the times for various attempts to short-circuit this function call, and the code for the various approaches.

   The actual function used by Amber is the slowest, partly because it has to check for non-Smalltalk objects to wrap them in `JSProxyObjects`. The best code is not completely clear, so we are currently using a simplified version of the `$recv` code which Amber uses – with that extra check removed. Additionally, it appears that there are numerous situations when we know that a value is not `undefined` (for example `self`) where we can bypass the call.

4.3 Instance Variables

As described in §2.3, the Amber developers decided to encode instance variables as fields prefixed with "@". Because the resulting name is not a valid Javascript name, accessing the variables has to use the indexing syntax rather than the dot syntax.

In figure 7 it is evident that there is essentially no difference for most of the engines. However, for Safari, there is a *very* significant difference. Therefore, we have changed the prefix of instance variables to `$_`.

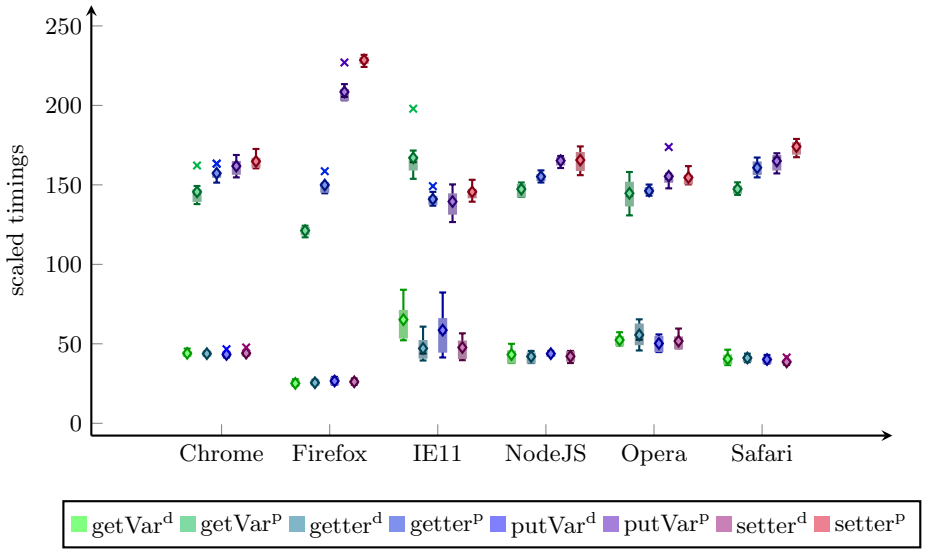

Figure 5: Direct versus proxy access to object fields

amber	`$amber(f).foo()`
eq2	`(f==null?nil:f).foo()`
or	`(f||(f==null?nil:f)).foo()`
or2	`(f||(f===undefined||f===null?nil:f)).foo()`
or3	`(f||(f===null||f===undefined?nil:f)).foo()`
or4	`(f||(f===null?nil:f===undefined?nil:f)).foo()`
recv	`$recv(f).foo()`
recvOr	`$recvOr(f).foo()`
orRecv	`(f||$recv(f)).foo()`

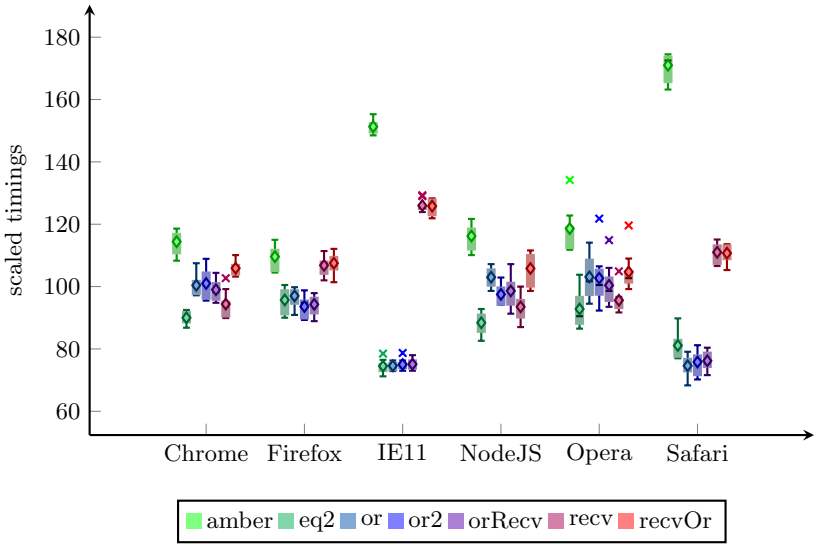

Figure 6: Recognizing `nil` values

always	`n+m`
alwaystes	`typeof m==="number"?n+m:m.adaptN(n,'+')`
send	`n.__plus_(m)`
sendtest	`n.__plusT_(m)`
test	`typeof n==="number"?n+m:n.__plus_(m)`
testtest	`typeof n==="number"?typeof m==="number"?n+m: m.adaptN(n,'+'):n.__plusT_(m)`

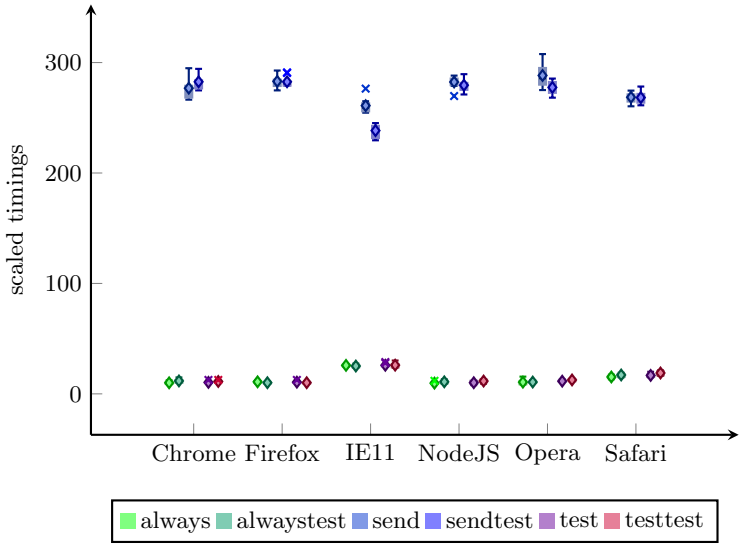

Figure 8: Optimizing Numeric Calculations

4.4 Numbers

The largest amount of code in most programs relates to numbers.

Amber sends messages (calls functions) for every numeric calculation. This is very expensive, and contributes to the amount of code that is written in native Javascript. When we recognize that values are numbers (similar to the Boolean test) we can replace a function call with a native Javascript operation. This can often be determined statically, either because the value is a numeric literal or because it is a reference to `self` in a method of class `Number`.

In figure 8 for all engines, the advantages of not calling the function are obvious. Amber does the `send` option. The "test" version of each pair adds a test to verify if the right-hand side is a number, and if not, Smalltalk semantics requires a call to `adaptToNumber:andSend:`.

5. AMBER AND CROSS-COMPILATION

Traditionally cross-compilation is only a moderately complex task.

One starts with a working compiler that runs on architecture X, and writes a revision of the compiler that runs on architecture X and generates code for architecture Y. Then compile the compiler itself with that revision and produce a compiler that runs on architecture Y. And the task is complete.

Amber was designed as an interactive compiler that modifies the methods in the running image as they are compiled. If the new architecture (or runtime environment) is incompatible with the current image the standard cross-compilation model fails – subtly or spectacularly.

For this project, an extra step had to be added to the cross-compilation process.

1. Revise the compiler to add a switch to prevent loading the code as it is compiled. Make note of the packages, classes, and methods compiled. Output only the components that were compiled, otherwise removing methods from existing classes fails. This stage has to remain compatible with the existing compiler and environment.

2. Revise that compiler to generate code for the new architecture. This still runs on the old architecture but doesn't update the running code.

3. Revise the support code for the new architecture and compile the compiler for the new environment.

6. RELATED WORK

There are numerous compilers targeting Javascript[8], and numerous Smalltalk compilers[10, 11, 3, 1, 13]. Obviously the most relevant previous work is the Amber[9] Smalltalk system upon which this was based.

7. CONCLUSIONS

This compiler is far from complete, but we are already seeing more efficient code being generated.

More importantly, the structure is in place and we have significant improvements envisaged.

Once it is more stable, we will suggest that Amber integrate portions of the code, though the degree of integration may be somewhat limited until the preponderance of browsers support the required features.

8. ACKNOWLEDGMENTS

Thanks to Nicholas Petton, Herbert Vojčík, and numerous others who are working on the Amber environment. This work would not be possible without building on their foundations.

References

[1] *Cincom VisualWorks Smalltalk*. URL: http://www.cincomsmalltalk.com/main/products/visualworks/.

[2] *ECMAScript 6 Rev 38 Final Draft*. Apr. 2015. URL: http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts.

[3] *Gemstone/S*. URL: http://gemtalksystems.com.

[4] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Don Mills, Ontario: Addison-Wesley, 1983. ISBN: 0-201-11371-6.

[5] *Kangax Browser Compatibility*. URL: https://kangax.github.io/compat-table/es6/.

[6] Ilya Kantor. *Prototypal inheritance*. 2011. URL: http://javascript.info/tutorial/inheritance.

[7] Ilya Kantor. *Pseudo-classical pattern*. 2011. URL: http://javascript.info/tutorial/pseudo-classical-pattern.

[8] *List of languages that compile to JS*. URL: https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS.

[9] Nicholas Petton, Herbert Vojčík, et al. *Amber Smalltalk*. URL: http://amber-lang.net.

[10] *Pharo Smalltalk*. URL: http://pharo.org.

[11] *Squeak Smalltalk*. URL: http://www.squeak.org.

[12] David Ungar and Randall B. Smith. "Self: The power of simplicity". In: *SIGPLAN Not.* 22.12 (Dec. 1987), pp. 227–242. ISSN: 0362-1340. DOI: 10.1145/38807.38828. URL: http://doi.acm.org/10.1145/38807.38828.

[13] *VisualAge Smalltalk*. URL: http://www.instantiations.com/products/vasmalltalk/.