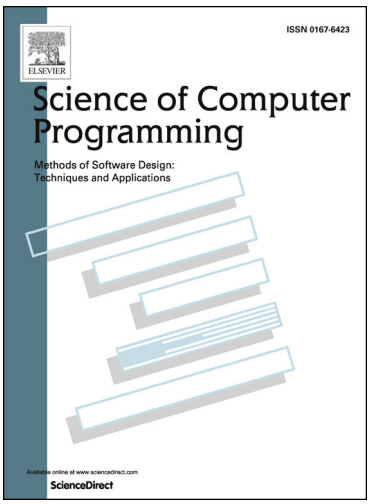
Accepted Manuscript



Test-Driven Development for Generated Portable Javascript Apps

Noury Bouraqadi, Dave Mason

PII: S0167-6423(18)30059-5
DOI: https://doi.org/10.1016/j.scico.2018.02.003
Reference: SCICO 2193

To appear in: *Science of Computer Programming*

Received date: 23 January 2017
Revised date: 5 January 2018
Accepted date: 26 February 2018

Please cite this article in press as: N. Bouraqadi, D. Mason, Test-Driven Development for Generated Portable Javascript Apps, *Sci. Comput. Program.* (2018), https://doi.org/10.1016/j.scico.2018.02.003

Highlights

- Produces fast, standard, Javascript code preserving Smalltalk semantics.
- Javascript debugging by leveraging the Pharo Smalltalk IDE.
- Final Javascript applications for deployment are fully standalone.
- Same test suite runs on multiple JS interperters to verify portability.

Test-Driven Development for Generated Portable Javascript Apps

Noury Bouraqadi[a], Dave Mason[b]

[a]*IMT Lille Douai, France*
noury.bouraqadi@mines-douai.fr
http://car.imt-lille-douai.fr/noury
[b]*Ryerson University, Canada*
dmason@ryerson.ca
http://sarg.ryerson.ca/dmason/

Abstract

With the advent of HTML 5, we can now develop rich web apps in Javascript that rival classical standalone apps. Nevertheless, developing in Javascript is still challenging and error prone because of the language's ambiguous semantics and quirks. In this paper, we advocate that the popular solution of using another language with better semantics and constructs is not enough. Developers should be provided with an IDE that eases the integration of Javascript libraries and enables testing an application across the many available Javascript interpreters. We introduce PharoJS[1], an infrastructure that allows Test-Driven Development (TDD) in Pharo Smalltalk of applications that ultimately run on a Javascript interpreter. PharoJS makes it possible to run interactive tests within the Pharo IDE, so as to fully exploit the debugging and development environment, while UI and libraries reside on the Javascript side.

Keywords: Test-Driven Development, Javascript, IDE, Crossplatform, Transpilation, Web development

1. Introduction

Javascript is one of the most ubiquitous languages in today's computing world. It is used for many applications such as web clients, web server-side applications,

[1] http://pharojs.org

as well as standalone apps on both handheld devices and desktops. On handheld devices, Javascript standalone apps rely on infrastructures such as PhoneGap, while on desktops they are based on infrastructures such as appjs, Electron, NW.js [1, 2, 3].

Nevertheless, developing in Javascript is still challenging and error prone[2] [4]. There is even a community web page on GitHub that lists the most quirky parts of Javascript, common pitfalls and subtle bugs[3]. Since Javascript is ubiquitous, these flaws have a dramatic impact on many projects.

To address these problems, one solution is to rely on IDEs with analysis tools to help detect and avoid Javascript pitfalls. However, most of the community effort is oriented towards having a language with better semantics. The ECMA association with the ECMASCRIPT standard aims at a progressive improvement of Javascript semantics. A more drastic approach is to develop in other languages, often new ones, and to use Javascript as an assembly language. The process of compiling to Javascript some code written in another language is called *transpilation*. This approach is favoured by the community. Indeed, it is adopted in 285 projects as reported on GitHub[4], involving major industry players such as Google (at least 6 projects including Caja, Dart and GWT) and Microsoft (at least 4 projects including TypeScript).

Another issue faced by Javascript developers stems from the existence of multiple actors providing different implementations of the ECMAScript standard. Implementing features from the standard is obviously a long process. Each team has a different pace and different priorities. So, at a given point in time most web browsers are only partially compatible. ECMAScript 5 was released in 2009, but as we write this, 8 years later, only the latest versions of Microsoft Edge and Mozilla FireFox web browsers claim 100% compatibility with the standard. Google Chrome is still not fully compatible since it scores 98%[5]. Things are worse regarding ECMAScript 6 released in 2015. No web browser is fully compatible yet (Microsoft Edge: 96%, Mozilla FireFox: 94%, Google Chrome: 97%)[6]. A workaround, is to use a tool such as BableJS[7] that converts ECMAScript 6 code to ECMAScript 5. However, this solution is in essence transpiling a different

[2] §AppendixA provides an example.
[3] See http://bonsaiden.github.io/JavaScript-Garden/
[4] See https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS
[5] See http://kangax.github.io/compat-table/es5/
[6] See http://kangax.github.io/compat-table/es6/
[7] https://babeljs.io/

language to Javascript. Another related problem is caused by the use of HTML and CSS. Again these technologies are defined by standards that are also not fully supported by all web browsers[8].

We argue that a solution for building applications based on web technologies while avoiding the above problems should meet the following requirements:

Modern OO language. Javascript should be replaced during development by a language, called the host language, which should be a *modern Object-Oriented Language* (OOL). We define a modern OOL as a programming language that provide features and constructs that are the norm in mainstream OO languages including class inheritance, trait inheritance [5], and lexical closures [6]. Applications developed in the host language are eventually converted to Javascript for production.

Full-fledged IDE. A full-fledged IDE for the host language is a must. It should minimize the developer's overhead to *safely* refactor code [7]. Safety means here that the new code has the same semantics as the original one. The introduction of errors is avoided by automating most of the process to achieve code transformation. Additionally, a full-fledge IDE should support debugging with live editing. That is, during debugging sessions, developers should be able to edit and replace code on the spot. This speeds up the bug fixing process, while providing developers with the full context where bugs occur. Lastly, IDEs should support optimisation and code quality. This requires deep software analyses, that are both static (e.g. class hierarchies, code quality metrics) and dynamic (e.g. method call stack, profiling metrics).

Portability Tests. Testing the portability of applications should be facilitated. A test infrastructure should run tests developed in the host language, over different environments (e.g. different web browsers). This allows the detection of potential incompatibilities between targeted Javascript interpreters or HTML renderers.

Integration Tests. Testing the integration of the code under development with third party Javascript libraries should be easy. This allows testing completed functionalities and detect potential integration issues early in the software life-cycle.

[8] See https://html5test.com/results/desktop.html

In this paper, we introduce PharoJS: a set of tools and libraries that support a modified Test-Driven Development (TDD) process to meet the above requirements and support our test-driven development process. We chose Pharo Smalltalk [8] as our host language. On the one hand, this decision is based on the fact that Pharo is a mature, pure, object-oriented language. On the other hand, Pharo has a rich IDE that provides features including: hot code swapping during debug, modifying live objects during debug, and support for code refactoring.

To deal with the portability test requirement, we have extended test definitions to include a list of targeted Javascript interpreters (e.g. different web browsers). Each test is run in two different ways on each targeted platform, which helps reveal integration and portability issues. The first time, only third party Javascript code runs on the Javascript interpreter. The code developed in Pharo is run in the Pharo run-time, so it can benefit from all features provided by Pharo and its IDE such as reference swapping, live editing/refactoring, and stack navigation to name a few. In the subsequent stage, the Pharo code is converted to Javascript and run on the Javascript interpreter, along with third party code. This allows for testing the app under the production conditions.

In §2 we evaluate the state of the art of Javascript-based application development. We show that the community is missing a solution that meets the above requirements. §3 gives an overview of our modified test-driven development process, and how it addresses the portability issue. Next, we present PharoJS in §4 and show how it meets the requirements for building Javascript applications. We illustrate development and testing of applications using PharoJS through a simple example. We also introduce the different kinds of proxies that support communications between Javascript interpreters and Pharo, during tests. §5 describes some of the key transpilation issues we addressed. §6 provides several benchmarks and compares PharoJS performance with other existing Smalltalk based solutions. §7 references some real-world applications built with PharoJS. Lastly, §8 draws conclusions and proposes future work arising out of the proposal described here.

2. State of the Art

2.1. IDE vs Separate Tools

There exist at least 285 languages that compile to Javascript[9]. The vast majority of these languages are experimental, and thus don't have a proper IDE. De-

[9]See https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

velopers have to code within some advanced text editor such as Atom[10], Brackets [11], Emacs[12], or Sublime Text[13]. These editors are flexible enough to allow developing plugins to support different languages. Thus, one can customize and extend such editors to provide basic refactoring operations (e.g. renaming) and code auto-completion for a given language. However, for testing and debugging, users have to rely on external, separate tools. Different tools might have different logics, flows, or ways to organize software artefacts. Thus, developers have to mentally bridge the gap between the different tools. This mental overhead results into a cognitive overload and fatigue that decrease productivity. Moreover, relying on multiple tools make some tasks hard if not impossible, as illustrated with Selenium [9].

Selenium is among the most widespread testing tools for web development. It supports running tests on multiple web browsers, and hence detect potential incompatibilities between different targets. To enable writing tests, Selenium provides an API for 10 languages including Javascript [14]. This allows writing tests in any host language that compiles to Javascript and provides a way to reference Javascript third party libraries (Selenium Javascript API in this case). However, Selenium performs only higher level functional tests that involve an interaction through the user interface. Beside omitting support for unit tests, Selenium does not provide any support to help developers track down and fix Javascript bugs (e.g. no break-points, no stack navigation, no live code editing).

With languages that have no dedicated IDE, developers have to rely on external debuggers provided by browsers such as Firebug [10], and Chrome DevTools [11]. Although such debuggers allow breakpointing, single-stepping, and examining of code and data, they operate on the generated Javascript code. Developers have to deal with the cumbersome activity of tracking back the original code to introduce fixes. An interesting workaround is to rely on Javascript source maps [12]. Source Maps, were initially introduced to allow web browser debuggers to retrieve and display the original Javascript code, while running a minified version that enables faster code loading over the network. This solution has been successfully applied to languages that compile to Javascript such as CoffeeScript [13] or TypeScript [14]. The map allows the Javascript engine to match up the Javascript

[10] https://atom.io
[11] http://brackets.io
[12] https://www.gnu.org/software/emacs/
[13] https://www.sublimetext.com
[14] See http://www.seleniumhq.org/about/platforms.jsp#programming-languages

code being evaluated with source developed in the host language.

Instead of using a set of different tools, another option adopted for many languages is to reuse a generic pluggable IDE. One way to find such an IDE for a Javascript based language is to analyze current trends in web development. The web development community maintains a list of the best IDEs[15]. At the time of writing, among the top 10 entries, 7 are not proper IDEs, but instead customizable text editors such as Atom or Emacs. Only 3 are actual IDEs that support different languages: WebStorm [15] (ranked 1st), Visual Studio Code [16] (ranked 2nd), and IntelliJ IDEA [17] (ranked 6th).

These IDEs and others existing ones such as NetBeans [18] or Komodo [19] support different languages and provide features for different steps in the software lifecycle. Nevertheless, they all lack support to ease the detection of potential incompatibilities between targeted Javascript interpreters. For example WebStorm allows debugging only on Chrome. Komodo allows debugging on both Chrome and Firefox. Visual Studio Code and NetBeans do much better since they allow debugging on various targets. However, tests can be run only on one web browser at a time.

Another limitation of the above IDEs is that the application code is *frozen* at debug time. One can only read the code being executed. To perform changes, developers have to stop debugging and go back to the edit mode. For decades, Smalltalk [20, 21] IDEs have provided support for live programming [22], allowing developers to change code on the spot and continue execution. This ability to change the code during execution greatly speeds up the debugging process. A version of this is actually implemented in text editors Brackets [23] (feature "Live Preview") and Light Table [24] (feature "Inline Evaluation"). To our knowledge, WebStorm is the only Javascript IDE that support live editing code during debug. But this is restricted to code run on the Google Chrome web browser.

2.2. *Review of Representative Solutions*

As stated above, there exist many languages that compile to Javascript (more than 285). There also exist many editors, IDEs, and test framework, which results in a combinatorial explosion of solutions (language + IDE + test framework). We have selected a subset for additional analysis with regard to the requirements we outlined in §1. As a baseline, we evaluate ECMAScript 5 and 6, with WebStorm,

[15] See blog post "What are the best javascript IDEs" http://www.slant.co/topics/1686/~what-are-the-best-javascript-ides.

one of the best IDEs for web development. For other languages, we selected Kotlin [16] and DART [25] because they are popular and backed by an important player: Google. We completed the subset with two existing solutions in the Smalltalk world: Amber [26] and SqueakJS [27].

	Language Constructs			IDE		
Language	Class Inheritance	Trait Inheritance	Lexical Closures	Live Edit on Debug	Portability Tests	Integration Tests
EcmaScript 5 + Webstorm	No	No	No	Only on Chrome	Specific to AngularJS	Yes
EcmaScript 6 + WebStorm	Yes	No	Yes	Only on Chrome	Specific to AngularJS	Yes
Kotlin + IntelliJ IDEA	Yes	No	Yes	No	No	Yes
Dart + WebStorm	Yes	Mixins	Yes	Only on Chrome	Specific to AngularJS	Yes
Amber	Yes	No	Yes	No	No	Yes
SqueakJS	Yes	Yes	Yes	Yes	No	Yes
PharoJS	Yes	Yes	Yes	Yes	Yes	Yes

Table 1: Comparison of Representative Solutions with PharoJS

Table 1 shows provides a summary of our evaluation of comparators. The three first columns compare language constructs. The three last ones compare IDE support for debug and testing.

ECMAScript 5 and ECMAScript 6 with WebStorm. Obviously, ECMAScript 5 is not satisfactory from a language perspective. ECMAScript 6 is a significant improvement in this regard, although it does not provide trait-based inheritance. Thanks to the WebStorm IDE, it is possible to use an actual debugger for integration tests, and even fix the code on the spot during debugging, but only with the Google Chrome browser. Regarding portability tests, they are only available for applications built on top of AngularJS. Indeed, developers can rely on the Protractor[17] test framework that allow running tests on multiple web browsers.

Kotlin with IntelliJ IDEA. From a language point of view, Kotlin provides high-level abstractions expected from a modern language, except for traits. Regarding tools, the recommended IDE for Kotlin is IntelliJ IDEA. By relying on a Kotlin to Javascript compiler, it can run integration tests that involve code developed in Kotlin and third party Javascript code. However, there is no support for running tests on multiple targets. Additionally, the application code is frozen during debugging, which slows down the development process, and developers lose the context where bugs are found.

[16] http://kotlinlang.org
[17] http://www.protractortest.org

DART with WebStorm. From a language stand-point, DART does meet our definition of a modern OO language. The only criticism at this level is that DART provides mixins, which are less compelling than traits since mixins lack the fine composition operators available with traits. Programs developed in DART can run on a dedicated Dart virtual machine that support hot reloading of code[18], and thus editing the code during debug. However, DART applications need to be converted to Javascript to run on web browsers[19]. WebStorm comes with DART support and allows live update of Javascript code. Therefor, live edit is possible during debug sessions - on the Google Chrome web browser. WebStorm also provides support for portability tests for AngularJS applications using the Protractor test framework.

Amber. Amber [26] is a subset of Smalltalk running in a web browser. As opposed to modern Smalltalk dialects, it has no support for traits. It features a compiler that converts Smalltalk code to Javascript, and allows integrating existing Javascript code. Amber, also provides an IDE implemented in itself. However, it has many limitations such as no hot-swap of code, a limited debugger, and no refactoring.

SqueakJS. SqueakJS [27] is a Javascript implementation of a Smalltalk VM, allowing execution of Smalltalk byte-code within the browser. This provides full modern Smalltalk semantics and abstractions, including traits, and allows interacting with the browser and 3rd-party Javascript libraries. However, tests can run only within the hosting web browser, hence excluding portability tests. While useful for exploratory programming in the browser, the performance of SqueakJS and the size of the required download will make it unusable for many applications.

PharoJS. We discuss PharoJS in the remainder of this paper. Here we only highlight the characteristics that leads us to choose Pharo. Pharo provides a rich OO language that includes support for traits and lexical closures. The Pharo IDE benefits from the reflective nature of the language, and enables evolving all facets of programs during debug (e.g. class structures, inheritance hierarchies, methods, object references). §4 presents how PharoJS extends Pharo to support portability and integration tests.

[18] https://2017.programmingconference.org/event/programming-2017-demos-live-development-in-dart

[19] Google abandoned Dartium which is a variant of the Chromium web browser that embeds the DART virtual machine.

3. Test-Driven Development for Integration and Portability

Our development process is based on Test-Driven Development (TDD), where tests serve as a guide to developers, as promoted by agile methodologies such as eXtreme Programming [28]. Figure 1 provides the TDD process highlighting steps specific to our approach.

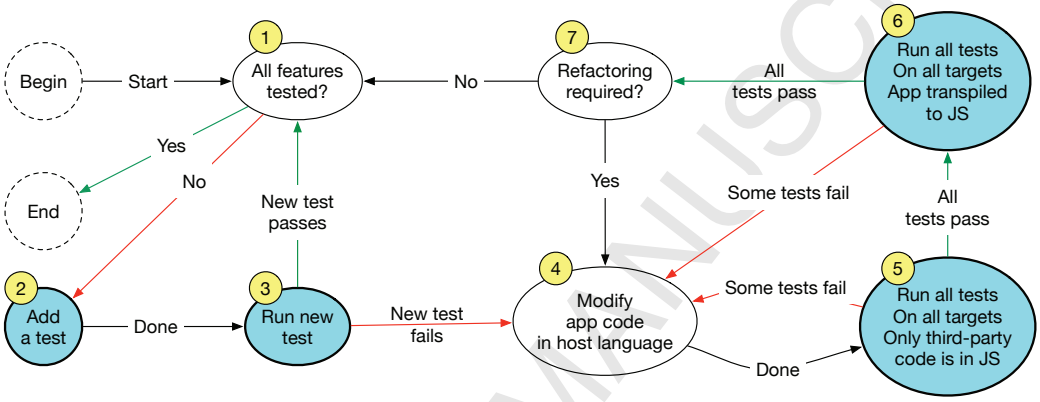


Figure 1: Test-Driven Development Process (Modified steps have coloured background)

When adding a test (step 2 of Figure 1), developers should define targeted platforms, i.e. targeted Javascript interpreters (e.g. different web browsers). The tests are run on every target platform (steps 3, 5, and 6). This allows detecting potential incompatibilities between platforms. It also allows testing features that are specific to some platforms. This is frequent in mobile development since different operating systems (e.g. iOS vs. Android) provide different services.

For a given platform, each test is run twice:

- In the first round of tests (step 5), only third party objects run on the Javascript interpreter. The code developed in the host language remains on the host language run-time.
- In the second round (step 6), the code in the host language is converted to Javascript and run on the Javascript interpreter, along with third party code.

The second round allows running the app under the same conditions to production. It also ensures that the host language execution model is faithfully replicated in the transpiled Javascript code.

We have introduced the first round to allow reusing an IDE built for a host language with a dedicated run-time. For example, Pharo runs on top of a virtual machine, and provides an IDE that includes different tools such as a debugger that supports hot code swapping at run-time. To benefit from the full power of this IDE the code has to run on the Pharo virtual machine. This is done in the first test round. So, developers can easily introduce break-points, step inside methods, inspect objects, and effectively fix bugs.

Since the run-time of the host language lives side-by-side with the Javascript interpreter, a communication bridge is required to connect the two worlds. Communication can go both ways: from the host language to the Javascript world and vice versa. This requires proxies of Javascript objects in the host world, as well as proxies of the host language objects in the Javascript world.

The middleware that supports these proxies must also allow controlling the Javascript interpreter from the host language and IDE. Running a test in the host language requires that the middleware allows us to remotely: start and stop the Javascript interpreter, load the communication support code, and evaluate any expression.

4. Developing Javascript Applications via Pharo

In this section, we introduce PharoJS, a solution to develop Javascript applications from within the Pharo Smalltalk [8] IDE. We show how PharoJS meets the requirements listed in §1.

4.1. The Host Language Pharo Smalltalk and its IDE

We directly address the two first requirements by choosing Pharo Smalltalk as our host language. Pharo is a free, industrial-grade implementation of Smalltalk. Smalltalk is a dynamic uniform OO language. Every entity is an object. Basic entities such literals (e.g. characters or numbers), as well as language constructs such as blocks, methods and classes are all objects, instances of classes.

The Smalltalk uniformity allows for a small set of concepts that result in a straightforward, clear syntax. For example, there are only 6 keywords, and only 5 possible operations: read a variable, write a variable, create a literal, send a message, and return.

Since Smalltalk is so dynamic, everything occurs at run-time, including programming. Because of this dynamicity and uniformity, it supports a set of advanced reflective capabilities that allows easy implementation of IDEs. For example, browsing the code can be achieved by introspection through a sequence of 4

messages: (1) send a message to the environment to get the list of packages; (2) send a message to a given package to get its classes; (3) send a message to a class to get the list of its methods; (4) send a message to a method to get its source code. Of course, the IDE uses these messages to support visual navigation of the code – including built-in classes.

The reflective nature of Smalltalk allows PharoJS to leverage portions of the Pharo Smalltalk IDE. The debugger is a cornerstone of the IDE. It allows navigating the call stack, as well as inspecting and modifying live objects. A unique feature of Smalltalk is reference swapping, which can be used during debugging to replace objects and code on the spot, and then continuing debugging with the new objects and the new version of the code. The new code is directly stored in the class it belongs to, so it continues to persist after the debugging session.

4.2. PharoJS Applications

To describe PharoJS applications, we introduce here a very simple example. Figure 2 provides the User Interface (UI) and the corresponding HTML and Smalltalk code. HTML tags that will be involved in the application's computations have identifiers. An index.js Javascript file is imported. Initially, this file does not exists. It is generated by the PharoJS tools upon running tests. The full content of this index.js file is Javascript code converted from Smalltalk code. During integration tests, it contains only middleware code to bridge Javascript objects with Smalltalk ones. During portability tests, it also contains the Javascript code of the application generated from the original Smalltalk code.

Every PharoJS application materializes as an object, in charge of creating and connecting other objects. In our example, the application materializes as an instance of class CounterApp. The application class is also used by the PharoJS transpiler as an entry point of the graph of classes to convert to Javascript.

4.3. Testing Applications in PharoJS

Figure 3 provides the full code to test both the integration with Javascript objects and the portability of the app. All test methods are implemented in the CounterTest abstract test class. This class contains most of the code, that is:

- the setUp method that retrieves the objects used by test methods
- test methods
- the targetPlatforms method that returns the list of targeted web browsers (typically: Safari, Google Chrome, and FireFox).

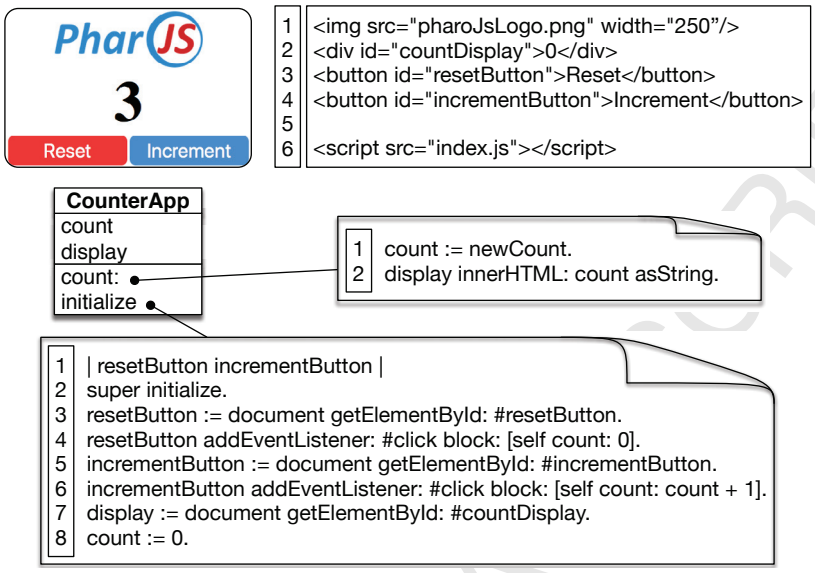


Figure 2: A Simple Counter App and its HTML and Smalltalk code

CounterTest defines an abstract method: newApp. This method is supposed to return an instance of the counter application.

The CounterIntegrationTest class redefines the newApp method to return a Smalltalk object. So, during integration tests, only DOM objects used by our app live on a web browser.

During portability tests, all objects including the one developed in Pharo Smalltalk are hosted by Javascript interpreters of the various targeted browsers. This is why the implementation of the newApp method of the CounterPortabilityTest class returns a proxy on a Javascript counter. The proxy is forged by converting an Smalltalk block to Javascript and evaluating it on a web browser.

The CounterPortabilityTest class also defines the appClass method. This method is used by the transpiler to track the application's classes that need to be converted to Javascript.

Based on the specified 2 test subclasses[20], 2 test methods[21], and 3 target Javascript interpreters[22], PharoJS performs 12 test runs. Indeed, for every concrete

[20] CounterIntegrationTest and CounterPortabilityTest

[21] testClickOnIncrementButton and testClickOnResetButton methods of CounterTest

[22] Web browsers listed in method targetPlatforms of CounterTest

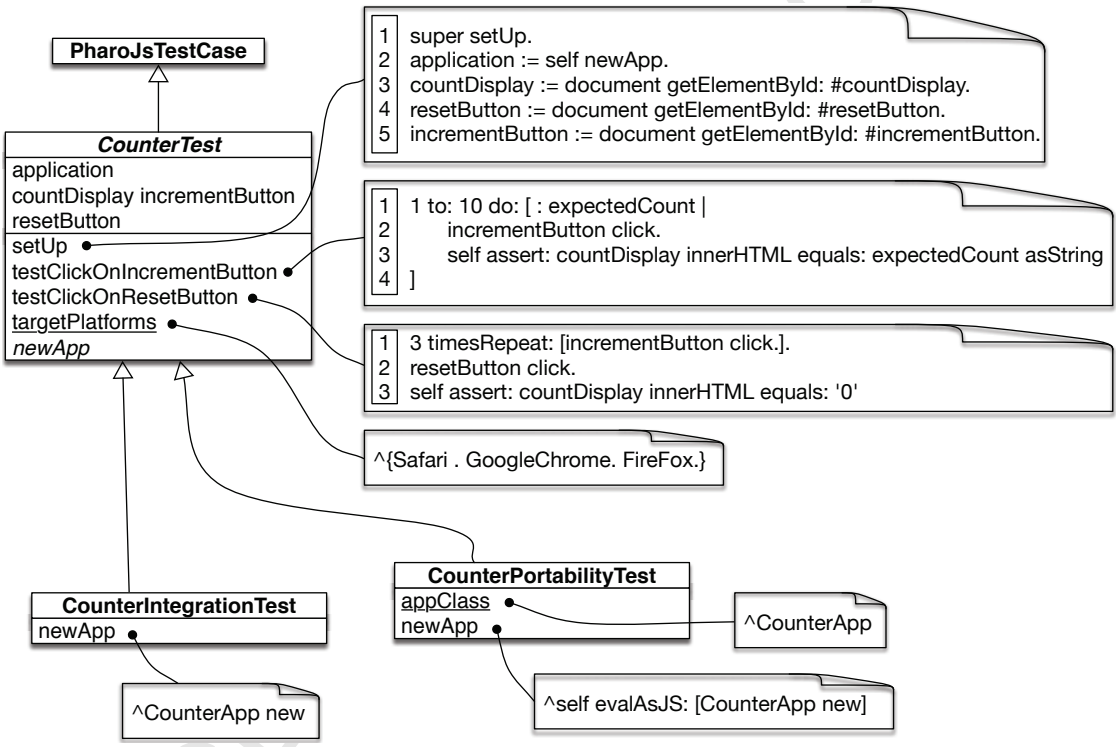


Figure 3: Integration and Portability Tests for a Simple Counter App

subclass of CounterTest (i.e. CounterIntegrationTest and CounterPortabilityTest), every test method is run 3 times: once for every targeted web browser. Developers can insert break points and execute any of these tests step-by-step. They can inspect Smalltalk objects as well as Javascript ones. Integration tests also allow stepping inside the counter application methods, executing them step-by-step, inserting break points. Developers also benefit from the full power of Pharo during integration tests. Since the application objects live in the Pharo run-time, one can change any part of the application code on the spot and see its consequences immediately during debugging sessions. Such changes can be adding/removing/-modifying methods, adding or removing instance variables (i.e. fields), or even change the class hierarchy. Obviously, all refactoring operations are also available while debugging integration tests.

To support tests, PharoJS includes a middleware that allow Pharo objects and Javascript ones communicate. This is achieved thanks to two kinds of proxies: *forward proxies* and *reverse proxies*. Forward proxies are placeholders of Javascript objects in Pharo. Conversely, reverse proxies are placeholders of Pharo objects in Javascript.

4.4. Forward Proxies: Referencing Javascript Objects in Pharo

This is the mechanism for the Pharo code to access objects living in the Javascript environment. We rely on Javascript global variables like document and window to provide initial entry points into the Javascript objects graph. For each Javascript global, PharoJS includes a ready to use proxy on the Pharo side. Besides, results of messages sent through these proxies to Javascript globals are converted to proxies in Pharo. Hence, we can create and access any Javascript object from Pharo. It is worth noting that primitive objects such as numbers and strings are passed by copy in this marshalling process.

Consider line 3 of setUp method in Figure 3. It shows the Javascript global object document referenced within a Pharo method. When running the test, the PharoJS middleware provides a proxy to the actual document object from the web browser. The message getElementById: is then sent over a web socket and leads to executing the getElementById() method from the core Javascript library. As a result, a DOM object (a DIV) is retrieved in the HTML of our app. The proxy created for this Javascript object by PharoJS is stored in the Pharo countDisplay instance variable (i.e. field). Thanks to this new proxy, we can send messages to the DIV in our tests (e.g. line 3 of testClickOnIncrementButton). Such proxies allow expressing in a very natural fashion code that involves both Javascript objects and

Pharo ones. Developers make no distinction between actual Pharo objects and Javascript ones that are accessed via proxies.

4.5. Reverse Proxies: Referencing Pharo Objects in Javascript

Referencing Pharo object in Javascript is mainly useful for handling callbacks from Javascript to Pharo. An example is provided in lines 4 and 6 of method initialize in Figure 3. Remember that Smalltalk blocks are objects. In our example, a block is provided as a parameter of message addEventListener sent to a Javascript button. Whenever the button is clicked, the web browser triggers Pharo to perform the block reset or increment the Pharo counter. This is achieved by creating a proxy in Javascript world, on the Pharo block.

Obviously, the Pharo object must be retained as long as there is a reference on the Javascript side, but no longer. There is a complication here, since there is no support for object finalization in Javascript. So, we have introduced a reference counting solution to deal with Pharo objects referenced by Javascript ones. Whenever a Pharo objet is passed to the Javascript world, it is stored in a dictionary at the Pharo side. So, it survives Pharo garbage collection thanks to this strong reference. On the Javascript side, a reverse proxy that references the Pharo object is created and its references are counted. When the count goes to zero, the Javascript reverse proxy sends a callback to remove the Pharo object from the dictionary, which allows for it to be garbage collected.

5. Transpilation to Javascript

Transpilation is a form of source-to-source compilation that aims to maximize semantics fidelity so that a program written in the original language (Smalltalk in our case) will perform the same when translated into the target language (Javascript in our case). Because Javascript is the only language supported on browsers for historical and compatibility reasons, it is also a very popular target language for transpilation. Smalltalk is a fortuitous source language, as it is possible to transpile it to fairly readable and high-performance Javascript.

5.1. Overview of the Transpiler

At the final stage of the PharoJS development process, the complete application is transpiled from Pharo Smalltalk to Javascript. The goal of the transpilation is to have the best possible fidelity to the original Smalltalk semantics, while interoperating smoothly with Javascript libraries and the DOM.

The PharoJS transpilation process ressembles the one introduced by Mario Wolczko to convert Smalltalk to Self [29]. Smalltalk Methods are converted to Javascript functions by traversing their Abstract Syntax Trees (ASTs). Smalltalk classes are converted to Javascript objects, that are connected in a way to preserve the parallel inheritance hierarchy (class inheritance and metaclass inheritance). However, our approach differs because of constraints inherent to Javascript (see Table 2) and our goal to allow Pharo developers to reuse third party Javascript libraries with a class-like mental model. In the remainder of this section, we discuss these issues and how we have addressed them.

Issue	*Solution*
Smalltalk is class-based while Javascript is prototype-based.	Use Javascript delegation for both Pharo inheritance and instantiation.
Any Smalltalk undefined variable points to an object (nil) in Pharo.	Dynamically replace undefined and null with nil object.
Smalltalk self pseudo-variable in block closures always refers to the receiver object that performs the method where the block was created.	Use a variable local in the method where the block is created instead of the pseudo-variable this.
Smalltalk block closures always answer a result.	Ensure Javascript functions that materialize Pharo blocks always return the result of the last expression.
Smalltalk block closures support non-local returns.	Use Javascript exceptions.
Smalltalk doesNotUnderstand: method	Generate trampoline methods for all possible selectors
Collision between methods with non-alphanumerical selectors (e.g. such as + or -) and Javascript operators	Use ASCII code.

Table 2: Major Issues Upon Transpiling Smalltalk to Javascript

5.2. Converting Classes

In Smalltalk everything is an object, including classes - which are unique instances of their corresponding metaclass. We are not aiming for complete metaprogramming, as many of those features are most relevant in building IDEs and other tools, and the PharoJS model is that these are available on the host Pharo system. However, we do want to support the principle of the class/metaclass parallel inheritance structure.

We implement Smalltalk classes as Javascript Function objects, which allows us to write library code in PharoJS that can be accessed by third-party code creating instances with `new`. The challenge with this structure is that the `__proto__` chain - by which inheritance is implemented in Javascript- has to be directly manipulated by PharoJS. The `__proto__` chain for the class/Function object (the putative instance of the metaclass and the source of the methods for the class) points to the class/Function object of the superclass. The `__proto__` chain for the class/Function object's `prototype` field (the source of the methods for instances of the class) points to the `prototype` field of the superclass class/-Function object.

5.3. Variables

In Smalltalk there are 5 categories of variables: local, instance variables, class instance variables, class variables, and global variables. All variables have the same name in Smalltalk and Javascript (except for those that conflict with Javascript reserved words). This facilitates interoperation with 3rd-party Javascript libraries. Because Smalltalk can have instance variables and methods with the same names, selectors are encoded as described in §5.5. Local and global variables have a natural mapping in Javascript, but the others require some explanation.

Instance Variables. When an instance of a class/Function is created in Javascript, an object is created and its `__proto__` field is initialized to the `prototype` field of the class/Function. This contains all of the instance methods encoded as described in §5.5. The created object initially has no other fields. When an uninitialized instance variable is referenced in that object, a Javascript `undefined` value is returned, which we treat as `nil` - the same as would happen in Smalltalk. Fields are created only upon the first assignment.

Class Instance Variables. In Smalltalk, the same thing happens when a class object is created from the metaclass. Because PharoJS doesn't attempt to provide full metaclass support, and because the `__proto__` chain points to the superclass object, we cannot depend on the uninitialized fields, as the superclass might already have the fields defined. Therefore PharoJS initializes all of the class instance variables to `undefined`.

Class Variables. Smalltalk class variables are inherited from superclasses and are accessible both from class methods and from instance methods. When accessing class variables from class methods, PharoJS treats class variables like instance

variables - they are in the class/Function object and not initialized. When accessing class variables from instance methods, PharoJS uses a specially-named accessor method which is defined in the `prototype` field of the class/Function where the class variable is defined. Because class variables must be inherited, assignment cannot simply attempt to store them in the object even in class methods, so instead a specially named setter method is created and a reference to it is created in both the class/Function object and in the `prototype`.

5.4. Converting Methods

To deal with methods, the PharoJS transpiler relies on two entities: a *converter* and a *generator* (see Figure 4). The converter implements a visitor pattern. It visits the Abstract Syntax Trees (AST) of each Smalltalk method and builds an AST of the equivalent Javascript code. The convertor allows for pluggable optimizations thanks to a collection of message conversions, organized according to the chain of responsibility pattern.

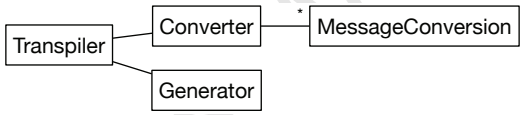


Figure 4: PharoJS Transpiler UML Class Diagram

AST conversion benefits from the fact that in Javascript all entities are objects - except for undefined and null - and support message sends. This allows for a straight-forward AST conversion, where we only take care of the following special cases:

- the receiver is undefined (discussed in §5.6)
- the selector is binary or keyword based (discussed in §5.5)

The Javascript code generator, also implements a visitor pattern. It traverses the Javascript AST and generates Javascript code that is compliant with ECMAScript 5. This separation makes it easy to adapt to Javascript interpreters, such as the different versions of the ECMAScript standard.[30]

5.5. Selectors

Pharo has three kinds of selectors (i.e. method names): unary, binary and keyword. A unary selectors is a sequence of alphanumeric characters beginning with a letter. An example is printString. It applies to methods/messages without any parameters. A binary selector is a sequence of non-alphanumeric characters. An example is >>. It is an infix form that applies to methods/messages that require only one parameter. Lastly, a keyword selector is a sequence of keywords. Each keyword is in turn a sequence of alphanumeric characters beginning with a letter and ending with a colon. An example is with:do:. It applies to methods/messages with one or more parameters. The number of keywords matches the number of parameters.

Although unary selectors could be used as-is to name Javascript functions, this would introduce potential collisions with field names. Indeed, by convention, Smalltalk fields and their read accessors have the same name. A collision may occur in Javascript, since both fields and methods are defined as slots. To prevent such collisions, and to minimize conflicts with 3rd-party Javascript libraries, we prefix all methods with a `$_`.

With both binary and keyword method names, we face the issue of forbidden characters in Javascript function names. For keyword method names, we simply replace each colon with an underscore. For binary method names, we replace each character with the digits of its ascii code followed by an underscore. Figure 5 gives examples that illustrate these rules.

abc printString	⇒	`abc.$_printString()`
abc@arg	⇒	abc._64_(arg)
abc >> arg	⇒	abc._62_62_(arg)
abc with: x do: y	⇒	abc._with_do_(x,y)

Figure 5: Examples of Smalltalk Selectors and their Javascript Counterparts

5.6. The Undefined Object

In Smalltalk an initialized variable references a special object named nil, the unique instance of the UndefinedObject class. In Javascript, prior to the first assignment, a variable holds the primitive special value undefined. Javascript also has

null which is another primitive value that developers may use instead of undefined. Although there is a subtle difference regarding the types[23], both are equivalent primitive values. Indeed, the expression null == undefined is true.

To ensure that nil is used instead of undefined and null on method dispatch, we need to replace undefined and null by nil. Since we want to allow the use of third-party Javascript code that might provide functions that answer these values, we dynamically replace undefined and null by nil. The PharoJS converter wraps the receiver of every message with a call to a function that performs the replacement – unless the receiver is known to be defined, such as `self`, numeric or string literals, or globals. This optimization can be seen in Figure 6.

5.7. Optimizations

To provide high-performance methods, Smalltalk compilers optimize certain methods - most particularly conditionals and loops - into in-line code. PharoJS performs the same optimizations, which makes many blocks go away, with significant performance improvement (see Figure 8). PharoJS also recognizes literal receivers to determine that arithmetic and string concatenation selectors can be inlined.

5.8. Non-local Return and Blocks

Smalltalk supports non-local returns whereby a return from within a block returns from the method in which that block is defined - regardless of where/when that block is evaluated. This simple feature enables creation of unique control structures using nothing more than methods and blocks. Unoptimized Smalltalk blocks are translated by PharoJS into Javascript functions, but Javascript has no such non-local returns. However Javascript does support try/catch and throwing arbitrary values.

Every method that includes an unoptimized block that references an instance variable or self initializes a variable called `$_self_$` to refer to the current object so that the nested function can reference it, since the Javascript pseudo-variable `this` will not refer to the same object within the nested function - see §AppendixA.

Every method that includes an unoptimized block with a return declares a variable `$_err_$` and initializes it to a unique value. Every non-local return sets this value to the return value and then throws the value. At the top level of such a

[23] typeof undefined results into undefined while typeof null results into object.

method there is a catch that compares the caught value to the `$_err_$` value. If the caught value is identical to the `$_err_$` value then it must have been thrown, so the value is returned from the method. If the caught value isn't identical, then it wasn't thrown by one of our blocks, so it must be either an actual exception, or a non-local return from some other method; in either case we re-throw the value. This optimization can be seen in Figure 6.

5.9. `doesNotUnderstand:`

Since Smalltalk relies on dynamic typing, checking that the class of receiver of a message does implement the invoked method is postponed to run time. Upon method lookup, if no method in the inheritance hierarchy meets the sent message, the type error is signaled. The Smalltalk virtual machine achieves this signaling by sending a `doesNotUnderstand:` message to the receiver. The unique argument is an instance of class `Message`. This is a reification of the faulty message.

Smalltalk root class (`Object`) provides a `doesNotUnderstand:` method that does simply raise an exception notifying the type error. However, the `doesNotUnderstand:` method can be redefined to perform arbitrary reflective computations [31]. We use this approach to support the integration of third-party Javascript code. More specifically, we rely on `doesNotUnderstand:` to read and set fields in Javascript objects.

The first problem we faced is that Javascript does not provide an equivalent to `doesNotUnderstand:`. Instead it throws an exception on a failed method call, whereas we need to catch it and possibly return a value. To make this work, we make sure that these exceptions can't happen by putting fall-back trampoline methods in the `Object.prototype` for every possible selector used in the system. Selectors that are discovered during transpilation are statically initialized at the beginning of the program, or before code is transferred to a running browser. Unknown selectors can also be sent using the Smalltalk `perform` method (and its parameterized versions) so trampolines are created dynamically. These fall-back methods create a Message object with the Smalltalk selector and the parameters, and call the `doesNotUnderstand:` method with the Message as the parameter.

There is also a fall-back `doesNotUnderstand:` that accepts unary and single-parameter keyword messages and treats them as getter/setter methods respectively for the corresponding field. If there is a field in the receiving object by that name that is not a function, then the field is returned or assigned. If the field is a function, then this is assumed to be a call to that method. This allows

```
thunkReady
   (self thunkOutputReady: foo ) ifFalse: [ ^ false ].
   inputs do: [: i |
       i ifNil: [ ^ false ].
       (i isAvailable: foo) ifFalse: [ ^ false ]
   ].
   ^ true

```

```
  function _thunkReady(){
    var $_err_$={},$_self_$=this;
    try{
      if(false==this._thunkOutputReady_(this.foo))
         return false;
      $asNil$(this.inputs)._do_(
         function(i){
            if(i==undefined)throw $_err_$=false;
            if($asNil$(i)._isAvailable($_self_$.foo))
               return undefined;
            else
               throw $_err_$=false});
      return true}
    catch(e){
      if(e===$_err_$)return e;
      throw e
    }})
```

Figure 6: Example method transpilation

accessing fields and methods in non-Smalltalk objects from the DOM or 3rd-party Javascript libraries. The object is also updated to have the getter/setter/method so that a subsequent call will access the field directly. If there is no field match, an error is thrown.

6. Benchmarks

PharoJS aims at targeting real world projects. The performance of the generated Javascript code is thus critical. In this section, we provide a series of benchmarks to show that PharoJS does exhibit satisfactory perfomance.

6.1. Methodology

We looked at key operations in the Javascript produced by PharoJS and created micro-benchmarks to compare the execution time in native Pharo, PharoJS, and Amber.

For each micro-benchmark, we have a loop of 1,000,000 iterations of a code block.[24] This is referred to as a run. The plots (e.g. Figure 8) reflect 10 runs of each micro-benchmark. The bars represent the average (mean) of the 10 runs. The box runs from the 1st quartile to the 3rd quartile and the whiskers show extremal points within 1.5 times that range of the median. For most engines and micro-benchmarks, the runs are very repeatable and show the median and mean as the same value.

The code generated by PharoJS assumes the ability to manipulate the `__proto__` chain, either by assignment to `__proto__` or by the function `setPrototypeOf()`. ES6 includes the `setPrototypeOf()` method, but some engines have supported assignment to `__proto__` for some time. Our generated code runs on all modern browsers: Chrome, Firefox, Safari, IE11/Edge, as well as NodeJS - in versions that have been available since mid-2014.

These experiments were not intended to be comparisons of browser performance, but rather to compare the code generated by PharoJS with the native Smalltalk code running on Pharo or SqueakJS, and the code generated by Amber. In the graphs, the simple browser names (Chrome, Firefox, Safari, NodeJS[25]) represent the browser running the code produced by PharoJS. CAmber and FAmber are Chrome and Firefox running the Amber generated code for the exact same

[24] Except for runs on SqueakJS, and the DeltaBlue and Richards benchmarks.

[25] even though NodeJS is not a browser, it is an important target for PharoJS

code. SqueakJS is a normal Squeak image running on a VM written in Javascript and running on Chrome. Finally, Pharo is the native code running on Pharo 5.

All the experiments were run on an otherwise idle Apple Macbook Pro, with 2.9GHz Intel i5 processor. To give the cleanest runs, the series of benchmarks was started each time on a fresh load of the browser. Because we are measuring relative performance for various code choices, rather than raw performance, any residual variability should not bear on the results.

It should be noted that tracking Javascript engine performance is tricky, as relative performance can change from release to release. In previous identical benchmark runs with different browser versions, we have observed quite different performance. We have tried below to identify significant performance differences.

Because of the range of execution times (often ranging over 2 orders of magnitude), the graphs are all semi-log graphs.

6.2. Results

In this section we discuss the results of the benchmarks and some of the code differences.

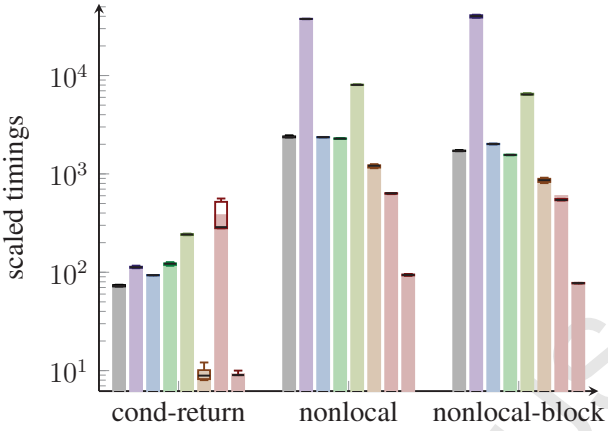
We include only a sample of the micro-benchmarks we have tested, and focus on the more realistic DeltaBlue and Richards benchmarks. Many more benchmarks are included in the conference version of this paper.[32]

Figure 7. shows the timing for conditional return from a method.

"cond-return" is special-cased by PharoJS and Amber, but PharoJS is currently a bit slower. NodeJS and Pharo execute an order of magnitude faster than Chrome, Firefox, and Safari.

PharoJS and Amber both handle non-local returns in a similar way and have similar performance (except for Chrome which is inexplicably an order of magnitude slower). However using an exception to handle non-local returns isn't nearly as fast as supporting it directly in the VM, so Pharo is 1.5 orders of magnitude faster, and SqueakJS is 0.5 orders of magnitude faster.

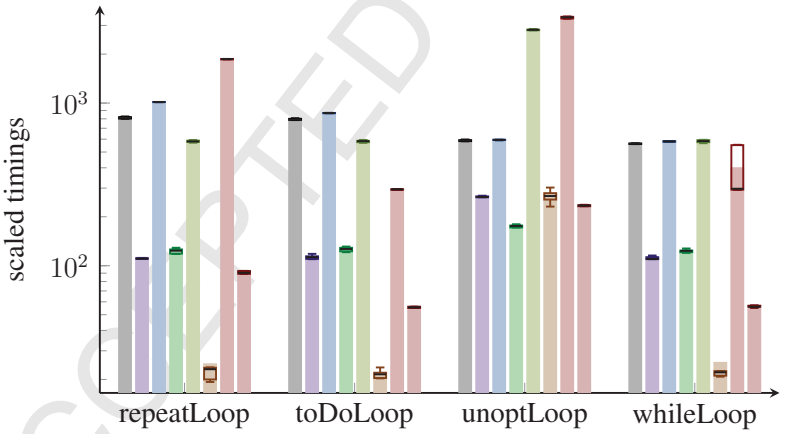
Figure 8. shows the timing for various kinds of loops. Except for "unoptLoop" which is intentionally unoptimizable, all the loops are special-cased by Pharo and PharoJS. The PharoJS code on NodeJS is actually faster than Pharo, and both are much faster than the other browsers. Amber does equally well on "unoptLoop" and "whileLoop" because they execute exactly the same code. "repeatLoop" and "toDoLoop" are slower because they are written using "whileLoop". PharoJS is significantly faster than Amber for "unoptLoop", and and an order of magnitude



CAmber Chrome FAmber Firefox Safari NodeJS SqueakJS Pharo

`cond-return`	`self ten>5 ifTrue: [^#yes]`
`nonlocal`	`self ten>5 ifTrue: [aBlock value]`
`nonlocal-block`	`self ten>5 ifTrue: aBlock`

Figure 7: Return expressions



CAmber Chrome FAmber Firefox Safari NodeJS SqueakJS Pharo

`repeatLoop`	`10 timesRepeat: [x:=x+1]`
`toDoLoop`	`1 to: 10 do: [:y| x:=x+y]`
`unoptLoop`	`b1 whileTrue: b2`
`whileLoop`	`[y<10] whileTrue: [y:=y+1. x:=x+y]`

Figure 8: Loop expressions

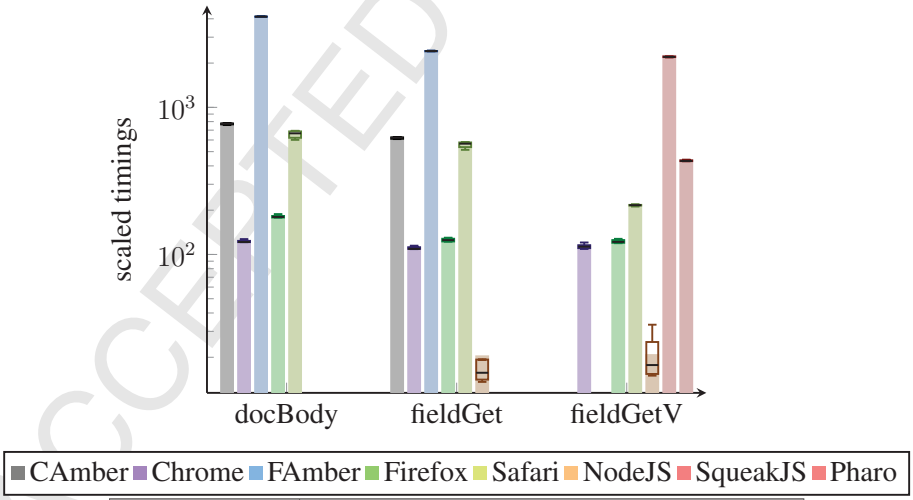
faster for the others. SqueakJS was timed running in Chrome, and is 1 to 1.5 orders of magnitude slower than PharoJS running on Chrome.

It's interesting that Firefox does significantly better on "unoptLoop" than the other engines. Safari is anomalously slow, as it has previously been comparable to the other web browsers.

Figure 9. shows the timing for accessing fields in objects. "docBody" doesn't make sense on NodeJS or Pharo which don't have a `document` object so it has been omitted from that test.

In Amber and PharoJS, these all work by the same mechanism: they call a function – that doesn't exist – on an object, and the call looks at the target object to determine if it has a corresponding field, which it sets or accesses. In Amber that is the end of it. In PharoJS we remember the action. We create a setter or getter function and insert it into the object, so that the next time the call is done, there *is* a function and it just executes normally doing the set or get operation.

If the field is non-existent or a functional value this doesn't work so the programmer must use the `instVarNamed:` mechanism to access the field. This is significantly slower on Pharo, but there is little performance degradation for PharoJS. Amber doesn't have `instVarNamed:` methods.

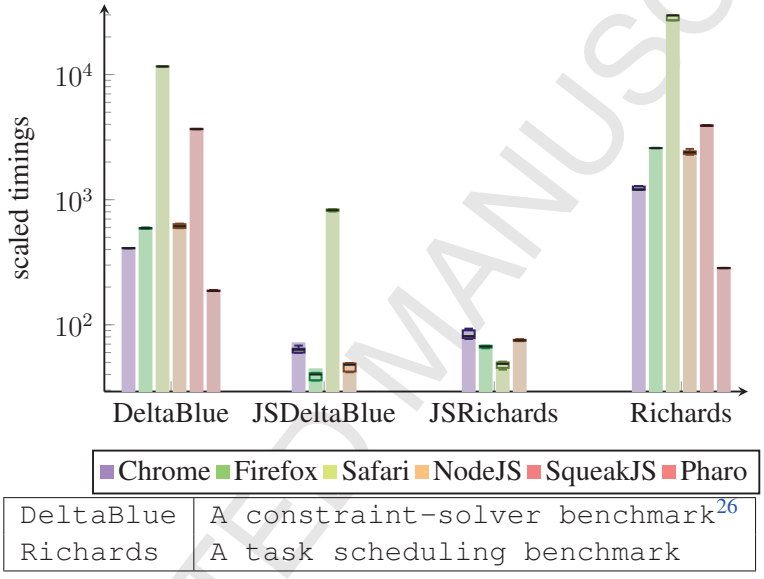


`docBody`	`document body`
`fieldGet`	`y := x index`
`fieldGetV`	`y := x instVarNamed: #index`

Figure 9: Field expressions

Figure 10. shows the timing for a couple of more realistic benchmarks. In addition to the PharoJS, SqueakJS, and Pharo timings, these also include timing for hand-written Javascript versions. PharoJS is generally 1.5 orders of magnitude slower than the Javascript versions.

The vagaries of Javascript timing are demonstrated by Safari that manages to turn in the fastest time of all on the Richards benchmark! SqueakJS also turns in a good time on the Richards benchmark, almost comparable to PharoJS.



`DeltaBlue`	`A constraint-solver benchmark`[26]
`Richards`	`A task scheduling benchmark`

Figure 10: Standard real benchmarks

7. Real World Applications

PharoJS has been publicly available since mid-2016, and has been used in several real-world applications, such as the ones listed on the PharoJS success stories web page[27].

The authors are aware of some details about 3 of them:

[26] Both downloaded from https://github.com/bonzini/smalltalk

[27] http://pharojs.org/successStories.html

7.1. A programming environment

programmingfortherestofus.com is a project developing a programming environment for citizen programmers. The environment has been entirely developed in PharoJS. It is the largest application developed to date with 167 classes and almost 700 methods. PharoJS exports a 2595 line Javascript file (420k that compresses to 67K).

7.2. A mobile application

The "Farmers Market" application[28] was developed with PharoJS and Apache Cordova. It runs on iOS and Android mobile devices. It is a mapping application that helps to locate nearby farmers' markets.

7.3. A teaching application

This application allows a presenter to present quizes and polls to an audience and displays the result back to the audience - similar to the clickers that some people use. It has a client application that runs on a mobile device or a laptop, and a server application that runs on NodeJS accessing a SQLite3 database.

8. Conclusion

While Javascript plays a prominent role in today's software, the language's ambiguous semantics makes development challenging and error prone. This is why there exist hundreds of languages that compile to Javascript. Their goal is to provide developers with a better language than Javascript, while still benefiting from the many web browsers and web views that allow targeting virtually any device.

In this paper, we advocate that having a high-level language with unambiguous semantics is not enough. Developers still need a full-fledged IDE. Moreover, they have to face two issues: the integration of third-party Javascript code, and the portability of the generated Javascript code. These two issues lead to unrelated bugs. Integration bugs can result for example from invalid assumptions on a Javascript library, or invalid references. Portability bugs are caused by disparities between Javascript interpreters that do not comply 100% with the ECMAScript standard.

[28] https://nootrix.com/farmers-market-app/

Our solution – PharoJS[29] – is built on top of Pharo Smalltalk. We chose Pharo because of its simple semantics, its high-level constructs (e.g. block closures, traits) and a rich IDE with unique features (e.g hot-code swapping during debug). PharoJS untangles integration and portability bugs by running the same tests under different conditions.

To uncover integration bugs, application objects are run on the Pharo virtual machine. A cross-language middleware allows Pharo objects to interact with core Javascript objects and third-party Javascript libraries. Objects running on the Javascript interpreter and involved in tests materialize in Pharo as proxies. Conversely, objects living on the Pharo side can be accessed through proxies from Javascript.

Portability bugs are revealed by running all objects on Javascript. A transpiler allows converting application code from Pharo Smalltalk to Javascript. The PharoJS middleware is still used to trigger actions and inspect both application and third-party objects that are at this stage all living on the Javascript interpreter.

This solution is almost overhead-free for developers, since tests are reused. All that is required is to provide a list targets that run the Javascript code. Additionally, our proposal integrates with a test-driven development process as promoted by agile methodologies. Developers can incrementally add new tests and reference new target platforms.

It is worth noting that although PharoJS supports the conversion of most Smalltalk code to Javascript, some features are still unsupported. Examples are concurrency, weak references, and some reflective operations such as accessing the execution stack (thisContext pseudo-variable), or swapping object references (the become: message). We plan to investigate in the future their conversion in a way that meets the current performance level exhibited by benchmarks we have conducted so far.

Other future work is to extend the Pharo IDE, to ease tracking down bugs related to portability. This can be nicely complemented with mapping Javascript code to the original Pharo code. Swapping Javascript code at run-time is also among planned PharoJS extensions.

AppendixA. Example of a Common Javascript Pitfall

A well known Javascript pitfall is related to callbacks, illustrated in Figure A.11. In this example, a method of the object person is passed to the function save as

[29] A pre-built ready to use version is available at http://pharojs.org/faq.html#prebuiltImage

a callback. However, the receiver is not attached to the method. Thus, when the method is performed by save, Javascript binds the this pseudo-variable to the global object window which obviously does not have the name slot.

```
1  var person = {                                   JS Code
2          name : 'John',
3          print : function(){console.log(this.name);}
4      };
5
6  var save = function(aCallback){
       /* Save data */
7      console.log("Done saving to database");
10     aCallback();
11 };
12
13 save(person.print);
```

```
Done saving to database                             Console
undefined
```

Figure A.11: Example of the Javascript callback problem

[1] M. Milani, B. Benvie, appjs: Build desktop applications.
URL http://appjs.com

[2] GitHub, Electron: Build cross platform desktop apps with web technologies.
URL http://electron.atom.io

[3] Community, Node-Webkit for desktop applications.
URL http://nwjs.io

[4] F. Ocariza, K. Bajaj, K. Pattabiraman, A. Mesbah, An empirical study of client-side javascript bugs, in: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, 2013.

[5] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behaviour, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2003.

[6] G. J. Sussman, G. L. Steele, Scheme: An interpreter for extended lambda calculus, Tech. Rep. AI Lab Memo AIM-349, MIT AI Lab (December 1975).

[7] Martin Fowler and Kent Beck and John Brant and William Opdyke and Don Roberts, ADDISON–WESLEY, 1999.

[8] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, 2009.
URL http://pharobyexample.org

[9] S. HQ, Selenium: Browser Automation.
URL http://www.seleniumhq.org

[10] Mozilla, Firebug: Web Development Evolved.
URL http://getfirebug.com

[11] Google, Chrome DevTools.
URL https://developers.google.com/web/tools/chrome-devtools/

[12] M. Foundation, Use a source map.
URL https://developer.mozilla.org/en-US/docs/Tools/Debugger/How_to/Use_a_source_map

[13] J. Ashkenas, CoffeeScript is a little language that compiles into JavaScript.
URL http://coffeescript.org

[14] Microsoft, Typescript. javascript that scales.
URL http://www.typescriptlang.org

[15] JetBrains, WebStorm Javascript IDE.
URL https://www.jetbrains.com/webstorm/

[16] Microsoft, Visual Studio Code.
URL https://code.visualstudio.com

[17] JetBrains, IntelliJ IDEA.
URL https://www.jetbrains.com/idea

[18] Oracle, Netbeans ide.
URL https://netbeans.org

[19] ActiveState, Komodo ide.
URL https://www.activestate.com/komodo-ide

[20] A. Goldberg, D. Robson, Smalltalk 80, Vol. The Language and its implementation, Addison-Wesley, 1983.

[21] F. Rivard, Smalltalk : a Reflective Language, in: REFLECTION'96, Edited by Gregor Kiczales, http://www.emn.fr/dept_info/neoclasstalk, San Franscico, USA, 1996, pp. 21–38.

[22] G. Wang, P. Cook, On-the-fly programming: Using code as an expressive musical instrument, in: Proceedings of the 2004 International Conference on New Interfaces for Musical Expression (NIME), 2004.

[23] Adobe, Brackets. a modern, open source text editor that understands web design.
URL http://brackets.io

[24] K. Inc., Light table. the next generation code editor.
URL http://lighttable.com

[25] Dart programming language specification - ecma-408 (2015).
URL https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-408.pdf

[26] N. Petton, H. Vojčík, et al., Amber smalltalk.
URL http://amber-lang.net

[27] B. Freudenberg, D. H. Ingalls, T. Felgentreff, T. Pape, R. Hirschfeld, Squeakjs: A modern and practical smalltalk that runs in any browser, SIGPLAN Not. 50 (2) (2014) 57–66. doi:10.1145/2775052.2661100.
URL http://doi.acm.org/10.1145/2775052.2661100

[28] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002.

[29] M. Wolczko, Self includes: Smalltalk, in: Ecoop'96 Workshop on Prototype Based Programming, 1996.

[30] Ecmascript (2015).
URL http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts

[31] F. Rivard, Smalltalk: a reflective language, in: Proceedings of REFLECTION '96, 1996, pp. 21–38.

[32] N. Bouraqadi, D. Mason, Mocks, proxies, and transpilation as development strategies for web development, in: Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies, IWST'16, ACM, New York, NY, USA, 2016, pp. 10:1–10:6. doi:10.1145/2991041.2991051.
URL http://doi.acm.org/10.1145/2991041.2991051