Challenges of Transpiling Smalltalk to JavaScript

Noury Bouraqadi[1,2,*,†], Dave Mason[3,†]

[1]IMT Nord Europe, Institut Mines-Télécom, Univ. Lille, Centre for Digital Systems, F-59000, Lille, France
[2]NOOTRIX, France
[3]Toronto Metropolitan University, Toronto, Canada

Abstract

Since Smalltalk has a great development environment, but somewhat limited deployment models, and JavaScript has ubiquitous deployment opportunities, but a relatively impoverished development environment it seems to be very advantageous to combine them. The aspiration is to "develop in Smalltalk, run in JavaScript". While there are a number of projects targeting transpiling Smalltalk to JavaScript and there are some interesting success stories, no solution fully realizes the potential for this model.

Currently, no description exists of how to approach a Smalltalk to JavaScript transpiler, and challenges that need to be overcome. In this paper, we discuss challenges to transpiling Smalltalk applications to JavaScript, and more specifically EcmaScript 6, based on insights we have acquired in the process of developing PharoJS.

Keywords

Transpilation, Smalltalk, JavaScript

1. Introduction

This paper focuses on challenges that need to be addressed for a successful *transpilation* of Smalltalk code to JavaScript. **Transpilation** is a contraction of the two words: *TRANSlation* and *comPILATION*. This is the conversion of a program developed in a source language into a functionally equivalent program generated in a target language.

Context. The number of projects aiming at building Smalltalk software that runs in the JavaScript run-time, demonstrates that this is an interesting topic for the Smalltalk community [1, 2, 3, 4, 5]. By achieving a Smalltalk to JavaScript mapping, one can benefit from the portability and the performance of the JavaScript runtime along with the efficient development experience of Smalltalk.

Nevertheless, even if there are some interesting success stories [6, 7, 8, 9, 10], no solution fully realizes the potential for the model: "develop in Smalltalk, run in JavaScript". Currently, no description exists of how to approach a Smalltalk to JavaScript transpiler, and identifying challenges that need to be overcome.

Requirements. A Smalltalk to JavaScript transpiler should support converting *any* valid Smalltalk code to an *equivalent* valid JavaScript code. That is, given the same input, both the Smalltalk code and the JavaScript code will produce the same output.

Challenges. Producing equivalent JavaScript code for *any* Smalltalk code requires mapping Smalltalk's syntax and its reflective kernel to JavaScript. These have *implicit* dependencies with Smalltalk's runtime and related concepts that should also be handled properly. While explicit dependencies are visible in the code through direct references, implicit dependencies are not. Examples of

IWST 2025: International Workshop on Smalltalk Technologies, July 1-4, 2025, Gdansk, Poland
*Corresponding author.
†These authors contributed equally.
noury.bouraqadi@imt-nord-europe.fr (N. Bouraqadi); dmason@torontomu.ca (D. Mason)
https://recherche.imt-nord-europe.fr/personnel/bouraqadi-noury/ (N. Bouraqadi); https://sarg.torontomu.ca/dmason/ (D. Mason)
0000-0001-6459-4934 (N. Bouraqadi); 0000-0002-2688-7856 (D. Mason)

Smalltalk run-time related implicit dependencies are: a class can have up to only 1 superclass, all entities are objects - instances of some class, and all variables initially reference the undefined object `nil`.

Although JavaScript was created as a prototype-based language, it has evolved to support a class-based programming style starting from version 6 of the ECMAScript standard. This reduces the gap with Smalltalk [11]. However, there are still several significant challenges to address when implementing a Smalltalk to JavaScript transpiler.

Focus. In this paper, we discuss challenges to transpiling Smalltalk applications to JavaScript. This analysis is based on insights we have acquired in the process of developing PharoJS [12, 13, 11]. We more specifically target EcmaScript 6 (ES6) as we showed in previous work [11] that ES6 support for classes allows for better performance of code generated by the transpiler.

Paper outline. First, in section 2 we catalog challenges that will be faced by any attempt to produce a high-fidelity transpilation from Smalltalk to JavaScript. While some items are adequately described there, others require more detail and are examined in the following sections (3 to 7): differences between low-level primitives and types in the two languages; messages; block closures; classes; and reflective operations. While some of these challenges are quite straightforward to address, others have significant nuances and limitations for achieving perfect alignment between the languages as summarized in section 8. Finally, section 9 draws some conclusions and identifies aspects that are beyond the scope of this paper.

2. Challenges Catalog Overview

We have identified 5 families of challenges that need to be addressed when transpiling Smalltalk to JavaScript.

Primitive Types and Literals. Smalltalk and JavaScript belong to the same family of Object-Oriented dynamic languages. Nevertheless, they differ even at the level of primitive types and literals.

Messages. Smalltalk messages can be reified, enabling support for custom dynamic type error handing by redefining the `doesNotUnderstand:` method. Beyond this reflective feature missing in JavaScript, there are other basic issues related to messages such as Smalltalk support for non-alphanumeric characters in selectors, or JavaScript math-like message priorities.

Block Closures. JavaScript anonymous functions only partially map to Smalltalk blocks. Thus, transpilers need to address challenges resulting from blocks supporting non-local returns and the fact that they always return a value.

Classes. Here are some class related concepts that only exist in Smalltalk: traits, class variables, shared pool variables, class initialization upon loading the class, as well as operations performed upon image startup and shutdown.

Reflection. Smalltalk provides multiple reflective features that are either not supported or only partially supported by JavaScript. For example, Smalltalk reified multiple concepts such as slots, execution contexts accessible via the `thisContext` pseudo-variable, as well as messages upon handling errors (`cannotInterpret:` and `doesNotUnderstand:`). Smalltalk also provides some reflective operations for which there is no direct mapping in JavaScript. Examples include pointer manipulation methods (`forwardBecome:` and `become:`).

In the following sections, we will expand on this catalog. Each family encompasses a set of challenges that we describe and illustrate with examples.

3. Primitive Types and Literals

Although both Smalltalk and JavaScript are dynamic object-oriented languages, they differ in the way they handle primitive types and literals. This includes, primitive types such as numbers, strings, and undefined objects. Smalltalk always relies on message sends (i.e. call methods) to perform any operation, including ones involving primitive types, literals, and operations such as basic arithmetic.

Differences between Smalltalk and JavaScript require transpilers to address the following challenges:

- JavaScript `undefined` and `null` are not objects.
- JavaScript has an impoverished numeric stack.
- Smalltalk automatically moves between small integers and large integers.
- Smalltalk supports fixed-point arithmetic.
- Smalltalk has literal symbols.

3.1. JavaScript `undefined` and `null` are Not Objects

In Smalltalk, every entity, including primitive types, is an object. In JavaScript, primitive types are automatically wrapped into full-fledged objects, so we can safely handle them just like in Smalltalk objects. The only exceptions are `undefined` and `null`.

Conceptually `undefined` and `null` are equivalent to Smalltalk's `nil` object. However, they are not objects and cannot be sent messages as objects. So, a transpiler needs to convert Smalltalk messages in a way that handles the case when the receiver (i.e. the object performing the method call) is `nil`. Furthermore the wrapped objects suffer a considerable performance penalty, so for optimal performance normal JavaScript operator syntax is preferred.

3.2. JavaScript has an Impoverished Numeric Stack

Smalltalk has a rich numeric stack including `SmallInteger`, `LargeInteger`, `Float`, `Fraction`, and `ScaledDecimal`, with smooth transitions among them. JavaScript, in contrast, has only numbers. JavaScript numbers are specified as IEEE 64-bit floating-point values, although internally most JavaScript engines do use small integers where possible. While the other Smalltalk numeric types could be emulated, it would be at considerable performance cost and smooth inter-operation is a daunting task.

3.3. Smalltalk Automatically Moves Between Small Integers and Large Integers

In Smalltalk, small and large integers are tightly integrated. On the one hand, when the result of an operation involving small integers is bigger than the largest small integer, we obtain a large positive integer. Similarly, when the result goes below the minimum small integer, we get a large negative integer. Conversely, if the result of an operation on large integers result in a valid small integer, that is what is returned.

On the other hand, $2^{53} = 9007199254740992$ is the end of the contiguous range of integers that can be exactly represented in JavaScript[1], so if we assigned `x=9007199254740992` then `x==x-1` is false, but `x==x+1` is true!

JavaScript has a class named `BigInt` to deal with large integers. However the conversion has to be done manually on some JavaScript runtimes. Besides, JavaScript has a different syntax for creating large integers. For example, `100n` creates an instance of a `BigInt` even if the number is small.

3.4. Smalltalk Supports Fixed-Point Arithmetic

Smalltalk allows writing arithmetic expressions with fixed-point numbers. Those are numbers with a fixed number of digits in their fractional part. They are instances of the `ScaledDecimal` class.

[1] there are many larger integers that can be exactly represented, but this is the end of the contiguous set

Smalltalk also supports a literal syntax for fixed-point numbers. For example, `0.128s + 1.0s` will result in `1.128s`, while with floating-point numbers, the result would be `1.1280000000000001`.

JavaScript has no syntax to express fix point numbers as literals. However, JavaScript numbers understand the `toFixed()` message that returns a string with the desired number of digits after the point. For example, the above Smalltalk operation on fixed-point numbers needs to be converted to the following JavaScript expression `Number.parseFloat((0.128 + 1.0).toFixed(3))`

3.5. Smalltalk has Literal Symbols.

Smalltalk symbols are strings that are **unique** in the whole Smalltalk image/program [14]. One creates a symbol by writing a sequence of alphanumeric characters preceded by a hash, as in `#odyssey2021`. So, the expression `#odyssey2021 == #odyssey2021` equals `true`. It means those are the same object, since in Smalltalk, `==` means identical. This is always true even if the symbol is created in different parts of a program or in different points in time. For comparison, two strings that are made of the exact same sequences of characters but created in different methods are different objects.

JavaScript does not have literal symbols despite having a builtin `Symbol` class. JavaScript `Symbol.for` is a function that creates unique `Symbol` associated with a particular string and `Symbol.keyFor` is a function that retrieves the string value. Unlike its Smalltalk counterpart, the JavaScript `Symbol` class is unrelated to the `String` class but is simply about uniqueness. Instances of JavaScript `Symbol` miss string operations such as concatenation or counting elements. Therefore, these possibly could be used to emulate the Smalltalk functionality, but smooth inter-operation with strings is difficult to achieve.

4. Messages

Smalltalk messages are equivalent to JavaScript method calls. A message has a selector, which is the name of the called method. Although we find the same concepts in both languages, transpilers should address the following differences:

- Non-alphanumeric characters in Smalltalk message selectors;
- JavaScript math-like message priorities;
- Smalltalk message cascading.

4.1. Non-Alphanumeric Characters in Smalltalk Message Selectors

Smalltalk binary and keyword messages selectors need special handling to be transpiled to valid JavaScript method names. Binary selectors are made of non-alphanumeric characters, such as `+` or `<`. Such characters are considered invalid in JavaScript method and function names.

Smalltalk message keywords, are sequences of alphanumeric characters ending with a colon (`:`). A keyword message selector is made of the concatenation of 1 or more keywords, including the colon characters. Consider the following Smalltalk code snippet. Two temporary variables `a` and `b` are declared and initialized with different integer arrays. The last line sends a message with selector `with:collect:` to the array referenced by `a`. The message has two arguments: `b` and a block i.e. a lambda function.

```
| a b |
a := #(98 45 66 73).
b := #(12 35 55 21).
a with: b collect: [:aElement :bElement | aElement + bElement ]
```

Just as with binary selectors, keyword selectors have characters that are forbidden in JavaScript method and function names. So, a transpiler must handle them properly. Simply skipping the forbidden

characters in the generated JavaScript is not an option. It might result in empty JavaScript identifiers when converting binary selectors or in collisions with other identifiers when converting keyword selectors.

4.2. JavaScript Math-Like Message Priorities

In Smalltalk all operations, including infix (typical math operations) are treated the same and sent as messages. When transpiling to JavaScript, the same approach could be used, i.e. all operations could be implemented as method calls, since JavaScript primitive types are automatically wrapped into full-fledged objects on a method call[2], but there is a significant performance cost, so it is important to use JavaScript expression syntax where possible.

Infix messages are always performed from left to right in Smalltalk, while JavaScript follows math priorities. So, an expression such as `2+2/4` will evaluate to `1` in Smalltalk while it evaluates to `2.5` in JavaScript. Thus, a transpiler must introduce parenthesis into the generated JavaScript code to ensure the consistency of transpiled math messages.

4.3. Smalltalk Message Cascading

Smalltalk allows sending several messages to the same receiver. For example consider the following Smalltalk listing. It shows a cascade of 4 messages (`loadStops`, `computeRoute`, `displayMap`, `listPointsOfInterest`). They all share the same receiver, which is a new instance of `TripPlanner`.

```
TripPlanner new
  loadStops;
  computeRoute;
  displayMap;
  listPointsOfInterest
```

There is no syntactic equivalent to this message cascading in JavaScript. A transpiler has to convert each message cascade to a sequence of basic independent messages. It also has to capture the value of the original target so the the same object is referenced, regardless of side-effects of the messages.

Transpiling the Smalltalk code for the trip planner example, would result in the following JavaScript. Note that we need to introduce variable `planner` to ensure all messages share the same receiver.

```
let planner = new TripPlanner();
planner loadStops;
planner computeRoute;
planner displayMap;
planner listPointsOfInterest;
```

5. Block Closures

Smalltalk blocks, which is short for block closures, are lambda functions. They are required to write any basic Smalltalk program. Blocks are for example used in conditionals and loops.

Transpiling Smalltalk blocks to JavaScript requires addressing the following challenges:

- Smalltalk blocks always bind the outer context;
- Smalltalk blocks always evaluate to some value;
- Smalltalk blocks support non-local returns.

[2] except, as mentioned earlier, `undefined` and `null`

5.1. Smalltalk Blocks Always Bind the Outer Context

Smalltalk blocks are by definition anonymous. They can refer to external variables (such as `self` the equivalent of `this` in JavaScript) from their context.

The following example provides an illustration. Class `A` defines a `block` method that answers a block that refers to `self`. The last line of the script creates an instance of `A`, obtains the block and evaluates it. It shows that the block can capture the receiver from its context via the pseudo-variable `self`.

```
A >> #answerToEverything
    ^ 42

A >> #block
    ^[Transcript show: self answerToEverything]

A new block value
```

To transpile a such code to JavaScript, we need to analyze the two ways to implements functions as provided by JavaScript:

- Functions defined using the `function` keyword can be anonymous. However, they do not bind the pseudo-variable `this`.
- Arrow functions are functions defined using the `=>` character. They are always anonymous.

The following example shows an arrow block successfully referencing the receiver, while an anonymous function raises a run-time error.

```
class A {
    answerToEverything() {
        return 42;
    }
    blockFromArrowFunction() {
        return () => {
            console.log(this.answerToEverything())
        };
    }
    blockFromAnonymousFunction() {
        return function () {
            console.log(this.answerToEverything())
        };
    }
}
let a = new A();
let block1 = a.blockFromArrowFunction();
block1(); // 42 is displayed
let block2 = a.blockFromAnonymousFunction();
block2(); // Runtime error
```

JavaScript arrow functions are more appropriate to transpile Smalltalk blocks than anonymous functions. However, they do not match all block features as shown in sections 5.2 and 5.3.

5.2. Smalltalk Blocks Always Evaluate to Some Value

When evaluating a Smalltalk block, unless it ends in a return (described in section 5.3), the result of the last expression is the value of the block. An empty block value is `nil`. To achieve the same behavior, transpilers must convert each block to a JavaScript arrow function ending with a return statement.

5.3. Smalltalk Blocks Support Non-Local Returns

Smalltalk blocks can be used to perform a return from the defining method. To illustrate this, consider a class `A` with the following two methods.

```
A >> #m1: aBlock
    Transcript cr; show: 'Before block value'.
    aBlock value.
    Transcript cr; show: 'After block value'

A >> #m2
    Transcript cr; show: 'Before m1'.
    self m1: [ ^ 42 ].
    Transcript cr; show: 'After m1'
```

Evaluating `A new m2` will send the message `value` to the block `[^42]` within `m1:`. Since the block returns, it will terminate both the execution of `m1:` and `m2`. As a result, the Transcript will display only the following:

```
Before m1
Before block value
```

Conversely, returns in JavaScript arrow functions only provide a value to the statement where they are performed. Consider the following class.

```
class A {
    m1(aBlock) {
        console.log("Before block value");
        aBlock();
        console.log("After block value");
    }
    m2() {
        console.log("Before m1");
        this.m1(() => {return 42});
        console.log("After m1");
    }
}
```

Evaluating `new A().m2()` will evaluate the arrow function within `m1`, then proceed with the execution of the last statements of `m1` and `m2`. The console will display all four lines:

```
Before m1
Before block value
After block value
After m1
```

With careful attention to detail, this non-local-return behavior can be replicated in JavaScript with `try`-`catch` and `throw`.

6. Classes

Transpiling Smalltalk classes to JavaScript requires dealing with the following challenges:

- Smalltalk has class variables;
- Smalltalk has pool variables;
- Smalltalk methods always have a return value;

- Smalltalk methods can have pragmas;
- Smalltalk primitive pragmas refer to the virtual machine;
- Smalltalk class initialization and startup/shutdown lists;
- Smalltalk supports class extensions;
- Smalltalk supports traits.

6.1. Smalltalk has Class Variables

These are variables shared among the defining class, its instances, its subclasses, and their respective instances. JavaScript has no support for class variables. Using `static` variables as discussed below only partially solves the issue.

For example, consider the Smalltalk code for a drawing app below. Class `Shape` defines a class variable `DefaultFillColor`, that can be changed using a class-side setter method `setDefaultFillColor:`. Each instance of `Shape` has a fill color, represented by *InstVar* `fillColor`. It can reference an arbitrary color. Message `resetFillColor` allows reverting it to the default color from class variable `DefaultFillColor`.

```
Object subclass: #Shape
        instanceVariableNames: 'fillColor'
        classVariableNames: 'DefaultFillColor'
        poolDictionaries: ''
        category: 'Demo'

Shape class >> #setDefaultFillColor: aColor
  DefaultFillColor := aColor

Shape class >> #getDefaultFillColor
  ^DefaultFillColor

Shape >> #resetFillColor
  fillColor := DefaultFillColor
```

An equivalent JavaScript code for this example is the following.

```
class Shape extends Object {
  static setDefaultFillColor(aColor){
    this.DefaultFillColor = aColor;
  }

  static getDefaultFillColor(){
    return Shape.DefaultFillColor;
  }
  resetFillColor(){
    this.fillColor = Shape.DefaultFillColor
  }
}
```

It is important to hardwire the reference to the class `Shape` in the static setter instead of using `this`. The rational is that class variables are shared with subclasses and subclass instances. Using `this` would instead create another static variable in the subclass.

It is worth noting that mapping class variables to JavaScript `static` variables is not enough. We need to avoid potential naming collisions with metaclass *InstVars*. Smalltalk metaclass *InstVars* are indeed mapped to `static` variables too.

6.2. Smalltalk has Pool Variables

Smalltalk allows having variables shared among different unrelated classes and their instances. These variables are defined in pools, that are referenced by classes. JavaScript lacks pool variables.

However, pool variables are a special case of class variables. Therefore, a solution exists based on an emulation similar to the one discussed in section 6.1.

6.3. Smalltalk Methods Always Have a Return Value

When source code for a Smalltalk method does not end with an explicit return statement, the compiler supplies byte code to return the receiver. This behavior needs to be replicated when transpiling Smalltalk code to JavaScript by appending a `return this` when there is no explicit return.

6.4. Smalltalk Methods can Have Pragmas

Smalltalk has a mechanism to provide annotations on methods, which are called "pragmas". Originally these were just mechanisms to access primitive operations such as arithmetic, and block evaluation (see section 6.5). They are now more broadly used to inform the compiler and reflective tools in Smalltalk.

But, JavaScript has no support for pragmas or any annotations for that matter. However, JavaScript objects structure can be extended with new properties. Since functions are objects, it is possible to extend their structure with pragmas as extra properties. This extension needs to be carefully performed to :

- distinguish pragmas for other properties, and
- avoid naming collisions with other properties

6.5. Smalltalk Primitive Pragmas Refer to the Virtual Machine

Some Smalltalk method bodies include a primitive pragma. This is a call to functionality of the Virtual Machine (VM), such as basic math operations, or input/output routines.

Analyzing how to transpile each Smalltalk primitive is beyond the scope of this paper. Suffice to say that there is no common solution to transpile all of them to JavaScript. Some, do not even have an equivalent in JavaScript. Examples are forcing garbage collection, swapping object identities (i.e. `become:`), suspending/resuming processes, or reading/writing compiled methods literals.

6.6. Smalltalk Class Initialization and Startup/Shutdown Lists

Smalltalk persists the content of RAM into a file called the image. It includes all objects, including classes. This is why the class initialization/finalization process is split in 2 parts. One is done upon loading/deleting classes to/from the image. The second is done upon each startup and shutdown, and often involves caches or resources outside the image.

Classes in the startup list perform some actions at the beginning of each session, right after the Smalltalk image is loaded in RAM. Classes in the shutdown list perform some actions at the end of each session, right before the Smalltalk image is unloaded from RAM.

Since JavaScript lacks an image, classes are reloaded upon each run. So, upon transpilation, we need to ensure useful startup operations are performed by the generated code for JavaScript class initialization. This can be done immediately after all class `initialize` methods have been called. We also need to provide a solution to perform shutdown actions in JavaScript.

It is worth noting that Smalltalk startup and shutdown actions are mostly related to image persistence. So, there is room for optimization by skipping operations that are useless in the absence of an image.

6.7. Smalltalk Supports Class Extensions

It is common practice in Smalltalk to introduce extra methods into classes from another project or library. Such methods can for example be introduced in core classes such as `Object`, to introduce application specific behaviors. This feature also supports polymorphism and helps in writing cleaner code. For example, it can be used to implement design patterns such Visitor, where many objects from different classes have to provide the same API [15].

In JavaScript one can create a function, then add it to an existing class. This practice called 'Monkey Patching' is discouraged in the JavaScript community. More importantly, JavaScript forbids creating functions that send messages to the `super` pseudo-variable.

Method name collision is yet another issue when extending existing JavaScript classes with methods from Smalltalk. The transpiler should avoid accidentally overwrite a symbol in an existing JavaScript class with a method from Smalltalk, so must make the Smalltalk name unique - such as adding an unusual prefix to Smalltalk names.

6.8. Smalltalk Supports Traits

Traits [16, 17, 18, 19, 20] were introduced as an alternative to multiple class inheritance. They enable code reuse in unrelated class hierarchies.

Just like a class, a trait can define methods and instance variables. Various operators allow developers to decide how to combine different, potentially conflicting traits into a given class.

While invented in the context of Smalltalk, traits have been adopted by other languages such as PHP and Python. However, JavaScript has no explicit support for traits. From a code reuse perspective, this is not an issue since methods defined in traits can simply be duplicated in all JavaScript classes that reuse them.

However this solution is not satisfactory from a reflection perspective. Traits have no existence in JavaScript, and one can not navigate the inheritance hierarchy and discover relationships between classes and traits.

For a full trait support, traits need to be reified in JavaScript. This is possible as shown by Van Cutsem and Miller [21]. With a such implementation, transpilation would require mapping Smalltalk traits to the JavaScript traits implementation.

7. Reflection

Reflection is the ability of a system to observe or modify its structure or its behavior [22]. Smalltalk has a rich Meta-Object Protocol, and many reflective abilities that we would like to account for when transpiling to JavaScript. Many concepts such as classes, methods, blocks, and contexts are reified in Smalltalk, materialized as classes that make up the Smalltalk kernel. These are critical to Smalltalk, because most implementations are live coding environments and the live coding tools are developed within the systems themselves.

Reflection is very limited in JavaScript. While there can be somewhat-live implementations, they are implemented outside the language itself. As in Smalltalk, JavaScript classes and methods are first-class objects and it is possible to parallel Smalltalk's class-metaclass inheritance hierarchies. What is more difficult (or impossible) is mapping Smalltalk's run-time reflection behavior such as contexts, `doesNotUnderstand:`, *et cetera*.

The following Smalltalk reflective features are the most difficult ones to address upon transpilation:

- Pharo Smalltalk reifies slots;
- Pharo Smalltalk classes define object formats, i.e. memory layout for their instances;
- Smalltalk reifies messages upon handling type errors;
- Smalltalk reifies execution contexts;
- Smalltalk and JavaScript have different solutions for intercepting method evaluation.

7.1. Pharo Smalltalk Reifies Slots

Slots are one of the new meta-level features introduced by Pharo Smalltalk [23, 24] that has no equivalent in JavaScript. Pharo extends class definitions to allow choosing the class for each slot. By default, the Smalltalk-80 instance variables behavior applies. But one can use or introduce subclasses of `Slot` to define custom access to individual slots. For example, some slots can be virtual - that is their values are computed upon access. Other examples are slots that keep the history of their values or notify observers upon changes.

7.2. Pharo Smalltalk Classes Define Object Formats

Developers can also define objects format, that is the objects memory layout to optimize or adapt it to their specific need. That is they define the structure of slots containers. For example, one can optimize the memory footprint of objects of some class to store slots in bytes or 32 bit words. Such layout would reduce the space occupied by boolean slots.

JavaScript lacks support for reified properties and their memory layout. Nevertheless, it allows some degree of property behavior customization through the meta operation `Object.defineProperty`. It allows hiding a property or making it read-only. It also supports the definition of meta-level `get()` and `set(newValue)` methods that intercept property reads and assignments.

7.3. Smalltalk Reifies Messages Upon Handling Type Errors

Both Smalltalk and JavaScript are dynamically typed languages. So, when an object receives a message that does not match any method, a runtime exception is thrown.

In Smalltalk, a type error occurs and the VM reifies the faulty message, and passes it as a parameter of a method named `doesNotUnderstand:`. The reified message is an instance of class `Message` that provides the message's selector as well as its arguments. The default implementation of this method as provided by the `Object` class, raises an exception. But subclasses can override it to introduce arbitrary code. This is often used as a technique to enable behavioral reflection by controlling message reception [25, 26].

Smalltalk flavors like Pharo that rely on the OpenSmalltalk VM have an extra layer for type error handling. "During the method lookup, if the method dictionary of one class in the class hierarchy of the receiver is **`nil`**, the VM sends the message `cannotInterpret:` aMessage to the receiver. Contrary to normal messages, the lookup for the method `cannotInterpret:` starts in the superclass of the class whose method dictionary was nil." [27]

In JavaScript the exception to signal type error is created by the VM. It is an instance of class `TypeError` that provides only explanatory messages intended to help a human debug the error. Given how frequently this is used in Smalltalk, support for a feature like `doesNotUnderstand:` is important. The only known way to efficiently handle this in JavaScript is to define, in `Object`, fallback methods for every possible message send. For this to work with `perform:` a method needs to be created at runtime if the perform selector does not already have a fallback method.

7.4. Smalltalk Reifies Execution Contexts

In Smalltalk, the state related to the execution of compiled code is materialized as instances of class `Context`. These objects can be accessed at runtime through a pseudo-variable named **`thisContext`** that references the run-time context of the code being executed. Reified contexts are used to implement continuations that are central to the Seaside web server framework [28].

A context references different objects such as the program counter, the method being executed, its receiver (**`self`**) and its arguments, as well as the current block closure if any. The context also references the parent context. So, it is possible to navigate the whole execution stack. This feature eases the implementation of debugging tools.

JavaScript lacks such a powerful construct. Nevertheless, it provides a pseudo-variable named `arguments` that references runtime objects that contain values of arguments passed to methods. An `arguments` object also references the method being executed through a property named `callee`. However, this property is deprecated.

7.5. Smalltalk and JavaScript Have Different Solutions for Intercepting Method Evaluation

Each Smalltalk class has a dictionary which maps selectors to methods. The Virtual Machine (VM) assumes that methods are instances of the `CompiledMethod` class. These objects include byte-codes to be executed by the VM as well as other meta-data used for reflective operations.

An effective technique to intercept method evaluation relies on so called *Method Wrappers* [29]. The OpenSmalltalk VM supports method wrappers. It enables intercepting method evaluations by replacing compiled methods in the method dictionary, by interceptor objects, i.e. method wrappers. The actual format of interceptor objects. The VM only assumes that they understand the `run: oldSelector with: arguments in: aReceiver` message. For example consider the following example of a Smalltalk method evaluation interceptor.

```
Universe >> #answer: anyQuestion
        ^42

Object subclass: #MethodEvaluationLogger
        instanceVariableNames: 'targetMethod'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Demo'

MethodEvaluationLogger >> #targetMethod: aMethod
        targetMethod := aMethod

MethodEvaluationLogger >> #run: oldSelector with: arguments in: aReceiver
        Transcript
                cr; show: 'Performing method: ', targetMethod selector;
                cr; show: 'Receiver: ', aReceiver asString;
                cr; show: 'Arguments: ', arguments asString.
        ^aReceiver withArgs: arguments executeMethod: targetMethod

| selector method interceptor |
selector := #answer:.
method := Universe compiledMethodAt: selector.
interceptor := MethodEvaluationLogger new.
interceptor targetMethod: method.
Universe methodDict at: selector put: interceptor.
Universe new answer: 'How many years before nuclear fusion?'.
```

In the first part of the script, class `MethodEvaluationLogger` is defined with instance method `#run: oldSelector with: arguments in:`. It logs information about the call, and evaluates the target method.

In the last part of the script, we first install an interceptor on method `#answer:` from class `Universe`. Then, we send the `#answer:` message to an instance of `Universe`. The interceptor logs the call information, then performs the target method and returns the result.

A similar behavior can be implemented in JavaScript using instances of the builtin `Proxy` class. Each proxy links a target object on which the interception should be performed to a handler object.

Interceptions are specified by methods supported by the handler.

For a proxy to intercept a method call, the target object should be a method. Additionally, the handler must understand the `apply(targetMethod, receiver, arguments)` method as in the following example.

```
class Universe{
    answer(anyQuestion){
        return 42;
    }
}

class MethodEvaluationLogger{
    apply(targetMethod, receiver, argumentsList){
        console.log("Performing method: ", targetMethod.name);
        console.log("Receiver: ", receiver);
        console.log("Arguments: ", argumentsList);
        return targetMethod.apply(receiver, argumentsList);
    }
}
const answerMethod = Universe.prototype.answer;
const proxy = new Proxy(answerMethod, new MethodEvaluationLogger());
Universe.prototype.answer = proxy;
(new Universe().answer("How many years before nuclear fusion?"));
```

8. Challenges Difficulty Summary

Challenges in the above catalog are more or less difficult. We measure this difficulty based on the existence of a solution, its completeness, and its efficiency. Using these criteria, we defined a challenge ranking scale, that has the following levels:

0. Solved: A solved challenge is a one for which there exist an efficient solution that addresses all cases.

1. Complex: A complex challenge is one for which there exist a solution, but it is is either difficult to implement or computationally demanding

2. Hard: A hard challenge is one for which we have a partial solution covering only a subset of possible cases.

3. Open: An open challenge is one for which there exist no known realistic solution except reimplementing a Smalltalk interpreter in JavaScript.

Table 8 provides a summary of all challenges discussed in this paper. Each challenge has a score based on its difficulty levels introduced above. The higher the score, the more difficult is the challenge.

9. Conclusion

Achieving perfect fidelity between Smalltalk and JavaScript via transpilation is impossible. If exact fidelity is a requirement, then the Smalltalk interpreter approach of SqueakJS[30] appears to be the only option. However, many of the more difficult aspects of fidelity are relatively rare elements of Smalltalk code, so with care many useful programs can be written in Smalltalk and deployed in JavaScript.

Of the challenges outlined in 2, the first four (**Primitive Types and Literals**, **Objects Structure**, **Messages**, and **Block Closures**) are the most critical for most programs. If a program doesn't use

Challenge	Difficulty Level	Comment
Primitive Types and Literals		
JS `undefined` and `null` are not objects	1. Complex	Computational overhead
JS has an impoverished numeric stack	2. Hard	Incomplete solution
ST automatically converts small integers to large ones	3. Open	
ST supports fixed-point arithmetic	1. Complex	Computational overhead
ST has literal symbols	2. Hard	Incomplete solution
Messages		
Non-alphanumeric characters in ST message selectors	0. Solved	
JS math-like message priorities	0. Solved	
ST message cascading	0. Solved	
Block Closures		
ST blocks always bind the outer context	0. Solved	
ST blocks always evaluate to some value	0. Solved	
ST blocks support non-local returns	0. Solved	Complex JS code
Classes		
ST has class variables	0. Solved	
ST has pool variables	0. Solved	
ST methods always have a return value	0. Solved	
ST methods can have pragmas	0. Solved	
ST primitive pragmas refer to the virtual machine	3. Open	
ST class initialization and startup/shutdown lists	2. Hard	
ST supports class extensions	0. Solved	
ST supports traits	2. Hard	Requires porting traits to JS
Reflection		
Pharo ST reifies slots	3. Open	
Pharo ST classes define object formats	3. Open	
ST reifies messages upon handling type errors	2. Hard	Incomplete solution
ST reifies execution contexts	3. Open	
ST vs JS interception of method evaluation	1. Complex	

Table 1
Summary of Challenges and Their Difficulty Levels

number classes other than `SmallInteger` and `Float`, they are also *relatively* straight-forward to implement, especially if execution performance is not critical. Fortunately, a great deal of engineering effort has gone into making JavaScript runtimes run very fast, so this will mask most of such issues.

The story is not so positive for the last two challenges, **Classes** and **Reflection**. There are several capabilities that can be routinely used in Smalltalk, that cannot effectively be implemented in JavaScript.

There are also several issues that are relevant but outside the scope of this paper. Most of these are doable, and will be described in future work:

- reusing JavaScript libraries (code + globals) in code transcribed from Smalltalk;
- Smalltalk and JavaScript run-time interoperability in production
- Smalltalk and JavaScript run-time interoperability in development, particularly supporting live-coding from the Smalltalk side - to the degree possible;
- transpiling Smalltalk to produce JavaScript libraries for consumption by 3rd parties.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] PharoJS: Develop in Pharo, Run on Javascript, 2023. URL: https://pharojs.org/, [Online; accessed 2023-11-08].

[2] CodeParadise, 2025. URL: https://github.com/ErikOnBike/CodeParadise, [Online; accessed 2025-05-14].

[3] SmallJS, 2025. URL: https://small-js.org/Home/Home.html, [Online; accessed 2025-05-14].

[4] Egg Smalltalk, 2025. URL: https://github.com/powerlang/egg, [Online; accessed 2025-05-31].

[5] Amber Smalltalk, 2025. URL: https://amber-lang.net/, [Online; accessed 2025-05-31].

[6] Apptivegrid: Digitise your processes today, 2023. URL: https://en.apptivegrid.de, [Online; accessed 2023-11-15].

[7] N. Bouraqadi, Plc3000: All-in-one solution for teaching factory automation, 2024. URL: https://plc3000.com/, [Online; accessed 2024-03-21].

[8] D. Mason, Covid-19 tracker, 2023. URL: https://covid.cs.ryerson.ca/compare/, [Online; accessed 2023-11-15].

[9] Pharojs success stories, 2024. URL: https://pharojs.org/successStories.html, [Online; accessed 2024-03-21].

[10] C. Haider, Covid-19 charts: Do not trust the numbers!, 2023. URL: https://covidcrt.uber.space, [Online; accessed 2023-11-15].

[11] N. Bouraqadi, D. Mason, PharoJS: Transpiling Pharo Classes to JS - ECMAScript 5 versus ECMAScript 6, in: S. Ducasse, G. Rakic (Eds.), IWST'23: Proceedings of the International Workshop on Smalltalk Technologies, ESUG, CEUR Workshop Proceedings, Lyon, France, 2023.

[12] N. Bouraqadi, D. Mason, Mocks, Proxies, and Transpilation as Development Strategies for Web Development, in: J. Laval, A. Etien (Eds.), IWST'16: Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies, ESUG, Association for Computing Machinery, New York, NY, United States, Prague, Czech Republic, 2016.

[13] N. Bouraqadi, D. Mason, Test-Driven Development for Generated Portable JavaScript Apps, Science of Computer Programming 161 (2018) 2–17.

[14] A. Goldberg, D. Robson, Smalltalk 80, volume 1 – The Language and its Implementation, Addison-Wesley, 1983.

[15] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1 ed., Addison-Wesley Professional, 1994.

[16] N. Schärli, S. Ducasse, O. Nierstrasz, A. P. Black, Traits: Composable units of behavior, in: Proceedings of European Conference on Object-Oriented Programming, volume 2743 of *LNCS*, Springer Verlag, 2003, pp. 248–274. doi:10.1007/b11832.

[17] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. P. Black, Traits: A mechanism for fine-grained reuse, ACM Transactions on Programming Languages and Systems (TOPLAS) 28 (2006) 331–388. doi:10.1145/1119479.1119483.

[18] O. Nierstrasz, S. Ducasse, N. Schärli, Flattening Traits, Journal of Object Technology 5 (2006) 129–148.

[19] A. Bergel, S. Ducasse, O. Nierstrasz, R. Wuyts, Stateful traits and their formalization, Journal of Computer Languages, Systems and Structures 34 (2008) 83–108. doi:10.1016/j.cl.2007.05.003.

[20] P. Tesone, S. Ducasse, G. Polito, L. Fabresse, N. Bouraqadi, A new modular implementation for stateful traits, Science of Computer Programming (2020).

[21] T. Van Cutsem, M. S. Miller, Traits.js: robust object composition and high-integrity objects for ecmascript 5, in: Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients, 2011, pp. 1–8.

[22] P. Maes, Concepts and Experiments in Computational Reflection, in: Proceedings of OOPSLA'87, ACM, Orlando, Florida, 1987, pp. 147–155.

[23] T. Verwaest, C. Bruni, M. Lungu, O. Nierstrasz, Flexible object layouts: enabling lightweight language extensions by intercepting slot access, in: Proceedings of 26th International Con-

ference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11), ACM, New York, NY, USA, 2011, pp. 959–972. URL: http://rmod.inria.fr/archives/papers/Verw11a-OOSPLA11-FlexibleObjectLayouts.pdf. doi:10.1145/2048066.2048138.

[24] P. Tesone, S. Bragagnolo, M. Denker, S. Ducasse, Transparent memory optimization using slots, in: IWST'18, Cagliari, Italy, 2018. URL: https://hal.inria.fr/hal-02565748.

[25] G. A. Pascoe, Encapsulators: A New Software Paradigm in Smalltalk-80, in: Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, 1986, pp. 341–346.

[26] S. Ducasse, Evaluating message passing control techniques in Smalltalk, Journal of Object-Oriented Programming (JOOP) 12 (1999) 39–44.

[27] M. Martinez Peck, N. Bouraqadi, L. Fabresse, M. Denker, C. Teruel, Ghost: A Uniform and General-Purpose Proxy Implementation, Journal of Object Technology 98 (2015) 339–359. doi:10.1016/j.scico.2014.05.015.

[28] S. Ducasse, L. Renggli, C. D. Shaffer, R. Zaccone, M. Davies, Dynamic Web Development with Seaside, Square Bracket Associates, 2009.

[29] J. Brant, B. Foote, R. E. Johnson, D. Roberts, Wrappers to the rescue (1998) 396–417.

[30] V. Freudenberg, SqueakJS, 2023. URL: https://squeak.js.org/, [Online; accessed 2023-05-13].