Semantics-Driven Interoperability between Scala.js and JavaScript

Sébastien Doeraene Tobias Schlatter Nicolas Stucki
École polytechnique fédérale de Lausanne, Switzerland
sebastien.doeraene@epfl.ch schlatter.tobias@gmail.com nicolas.stucki@epfl.ch

Abstract

Hundreds of programming languages compile to JavaScript. Yet, most of them fail, at one level or another, to provide satisfactory interoperability with JavaScript APIs. This is limiting, as interoperability is at least required to manipulate web pages through the DOM API, but also to use the eco-system of existing JavaScript libraries. This paper presents the interoperability features of Scala.js, which solves the shortcomings of previous approaches. Scala.js offers a separate hierarchy of JavaScript types, whose operations have semantics borrowed from ECMAScript 2015. The interoperability features are complete with respect to ECMAScript 2015, save for two exceptions which are still being worked on. This allows Scala.js programs to perform any operation that an ECMAScript program could do, thereby guaranteeing that they can talk to any JavaScript library.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]

Keywords language interoperability, object-oriented, functional

1. Introduction

Nowadays, there are literally hundreds of programming languages compiling to JavaScript. Some languages only add simple syntactic sugar, such as CoffeeScript [2], or add static types to JavaScript, like TypeScript [15] and Flow [7]. Other languages were created from scratch to target JavaScript, among which Dart [10], Haxe [8], PureScript [18] and Elm [5]. Finally, several languages designed for other runtimes, such as the JVM or the CLR, were ported to also run in JavaScript runtimes, e.g., GWT [11], ClojureScript [1] and FunScript [9]. All of them share one crucial requirement: being able to talk in some way or another to JavaScript APIs. At the very least, invoking the DOM API is required to manipulate web pages, but we also want to be able to leverage the eco-system of existing JavaScript libraries.

When it comes to interoperability, we can separate the languages compiling to JavaScript into two main categories: languages with the same run-time semantics as JavaScript, and languages with different run-time semantics (such as compile-time overloading).

In the first category, we find languages such as CoffeeScript, TypeScript and Flow. In those languages, talking to JavaScript APIs is easy, since there is virtually no impedance mismatch between the languages. At most, it requires defining type definitions for JavaScript APIs to be able to call them easily, as is the case in TypeScript.

In the other category, we find all the languages that either offer more powerful language abstractions (e.g., PureScript, Elm) or support multiple target runtimes (e.g., ClojureScript, GWT). These languages all fail, at some level or another, to provide satisfactory interoperability with JavaScript APIs, as we will see in Section 2. Shortcomings range from the inability to handle object-orientation, overloading, or higher-order functions, to requiring knowledge of the implementation details of the compiler. If a language misses some interoperability features, there are some JavaScript libraries that it *cannot interact with*. Usually, this is worked around by writing “bridge” JavaScript code working as an adapter for the library, reexposing its functionality with a subset of the JavaScript language features understood by the program.

In this paper, we present the interoperability features of Scala.js [19], a dialect of Scala compiling to JavaScript. When designing Scala.js, interoperability with JavaScript has been the number one priority. An early design [3] was no better than the state of the art (it suffered from issues similar to those we discuss in Section 2), and prompted us to research a different approach to interoperability. While part of the second category of languages, Scala.js now solves the shortcomings of previous approaches to interoperability. It does so with what we call a semantics-driven approach: provide Scala.js language features for all the run-time semantics offered by the ECMAScript 2015 language, so that Scala.js

This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

SCALA’16, October 30–31, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4648-1/16/10...
http://dx.doi.org/10.1145/2998392.2998404

programs can literally do everything that ECMAScript programs can do. This guarantees that Scala.js programs can talk to *any* JavaScript library. At this stage, the semantics for two ECMAScript constructs–modules and `new.target`–are still being worked on, which means this guarantee is not actually provided by the currently implemented system, but will be as soon as they are addressed.

Contributions

- We extend the semantics of Scala with an additional type hierarchy of JavaScript types in Section 3, which belongs in the same type system but has different run-time semantics, defined in Section 4. Both at call site and definition sites, the semantics of operations on these types are borrowed from the semantics of ECMAScript constructs. Our interoperability features are complete with respect to ECMAScript 2015, save for two exceptions, which we discuss in Section 4.4.
- We provide a solution for interoperability with overloaded methods, in particular in the presence of inheritance and overriding, which has not been addressed in any practical nor theoretical system before. We reconcile the run-time overloading dispatch semantics of JavaScript with the compile-time overloading constructs of Scala, both at call site (Section 4.2.1) and definition site (Section 4.3.3).

2. Motivation

In this section, we present some shortcomings of previous approaches to interoperability with JavaScript. We give most examples in GWT and ClojureScript, because they are, in our opinion, among the languages providing the best interoperability features. Their few shortcomings are the most challenging ones to tackle. Note that these shortcomings are a property of the current design of interoperability features in these languages, and not fundamental limitations. Future versions of ClojureScript or GWT could address those concerns, possibly with the approach described in this paper.

Overloading Although JavaScript technically does not have overloading as a language feature per se, the term "overloading" is commonly used in JavaScript–including in its specification, e.g., [4, §20.3.2]–as referring to run-time dispatch based on run-time tests, both for the actual number of arguments provided at call site (arity-based overloading) and for the types of the actual arguments (type-based overloading).

Consider the following JavaScript code[1], which uses the overloaded method `html` of jQuery both to read and write.

```
const list = $("#list");
const oldHTML = list.html();
list.html(oldHTML + "<li>New elem</li>");
```

In a language whose interoperability features cannot express overloading, such as GWT in their latest version [12], calling such an API is not directly possible. Workarounds include a) writing a bridge JavaScript library that exposes `getHTML()` and `setHTML()` separately, or b) using two different types for `list` (one declaring the getter, the other the setter, with an explicit cast in between). Neither workaround is satisfactory. Besides, they do not really allow a GWT program to *implement* such an API, to be consumed by another JavaScript module. It is impossible to write a class exposing an overloaded method to JavaScript, as mentioned in Section "Caveats & Special cases" of the GWT interoperability specification [12].

GWT interoperability does not support overloading because there is a major *semantic mismatch* between compile-time overloading dispatch in Java and run-time overloading dispatch in JavaScript. In fact, to the best of our knowledge, the only languages that correctly handle overloading in their interoperability are those in which overloading has run-time dispatch semantics to begin with, such as ClojureScript.

Object-Orientation Consider the following JavaScript code using Phaser [13], a game development library. It creates a very simple game state that draws a triangle on the screen.

```
class GameState extends Phaser.State {
  create() {
    this.graphics =
      this.game.add.graphics(0, 0);
    this.graphics.beginFill(0xFFD700);
    this.graphics.drawPolygon(
        [50, 0, 100, 100, 0, 100]);
    this.graphics.endFill();
  }
}
const game = new Phaser.Game(
    100, 100, Phaser.AUTO, "container");
game.state.add("game", new GameState);
game.state.start("game");
```

Implementing the same functionality in ClojureScript is not possible, because ClojureScript does not expose any interoperability feature to create classes. As long as `Phaser.State` is declared as an ECMAScript 5.1 constructor function (and not as an actual ECMAScript 2015 class), a workaround is to create `GameState` as a function, then manipulate its prototype the old-fashioned way:

```
(defn GameState [] ...)
(set! (.-prototype GameState)
      (new (.-State js/Phaser)))
(set! (.. GameState -prototype -create) (fn []
  (this-as this
    (let [graphics (.graphics
          (.-add (.-game this)) 0 0)]
      (set! (.-graphics this) graphics)
      (.beginFill graphics 0xFFD700)
      (.drawPolygon graphics
        (array 50 0 100 100 0 100))
      (.endFill graphics)))))
```

However, this workaround would stop working if Phaser migrates `Phaser.State` to an actual ECMAScript 2015

[1] All our JavaScript examples assume ECMAScript 2015 [4], which is the latest standard as of this writing.

class. Indeed, it is not possible to extend an ES 2015 class from an ES 5.1 constructor function.[2]

Generics and Primitive Types GWT also has a more subtle issue in the previous example. The method `drawPolygon` takes an array of numbers as parameter. It would be tempting to declare the `JsType` for `Phaser.Graphics` as follows:

```
@JsType(namespace="Phaser", isNative=true)
class Graphics {
  native void beginFill(double color);
  native void endFill();
  native void drawPolygon(JsArray<Double> ps);
}
```

GWT's interoperability specifies that the `double color` will be seen by JavaScript as a number. However, due to auto-boxing in Java, `JsArray<Double>` is not a JavaScript array of numbers. It is a JavaScript array of instances of `java.lang.Double`, which Phaser cannot understand. Declaring a JavaScript array of numbers requires a separate, non-generic class. In general, generics cannot be instantiated to primitive types to represent JavaScript data types.

This time, the semantic mismatch is about boxing: Java boxes primitive values when they enter a generic context, whereas JavaScript keeps primitive values in all contexts.

Conclusion As we have shown in this section, existing interoperability solutions have severe limitations. When some constructs available to JavaScript applications are impossible to reproduce in a source language, there are JavaScript libraries that *cannot be used* by applications in that language. It is therefore important to design interoperability features that avoid this situation. This can only be achieved if those features allow to reproduce any behavior that could be achieved in JavaScript, i.e., if they cover "all of JavaScript". We will make this criterium more precise in Section 4.4, but will first go through the design of interoperability in Scala.js.

3. Scala Types and JavaScript Types

As hinted in the previous section, the main obstacle to good interoperability is a mismatch between the run-time semantics of two languages. A typical problem, for statically typed languages, is overloading. On the one hand, overloading in JavaScript is resolved at run-time. On the other hand, statically typed languages, among which Scala, resolve overloading at compile-time.

How can we address the semantics mismatch in Scala.js? Brutally changing that aspect of the Scala semantics when compiling to JavaScript is not acceptable, as it would break valid programs. Our solution to this problem is the following: do not shy away from the semantics mismatch, but rather acknowledge its existence, and encode it into the type system. We do this with a separate hierarchy of *JavaScript types*, rooted in the type `js.Any`, a third subtype of `Any` (beside `AnyVal` and `AnyRef`, which are Scala types). Unlike Scala types, JavaScript types have *JavaScript semantics*.

For example, recall the overloaded `html()` method of jQuery. We can type this API using a JavaScript trait in Scala.js. For the purposes of this paper, traits are similar to interfaces in Java.

```
trait JQuery extends js.Any {
  def html(): String
  def html(newValue: String): Unit
}
val list: JQuery = createSomeJQuery()
val oldHTML = list.html()
list.html(oldHTML + "<li>New elem</li>")
```

Because `list` has a JavaScript type, calling `list.html()` is intuitively equivalent to the corresponding JavaScript code, i.e., it looks for a property named `html` in the prototype chain of `list`, checks that it is callable, and calls it with `list` as value for `this` and zero argument. Similarly, `list.html(<expr>)` calls it with one argument, which is the result of evaluating `<expr>`. We will make the semantics of method calls more precise in Section 4.2.1.

When calling from Scala.js to *native* JavaScript code, this might seem obvious. The call `list.html()` cannot do anything but resolve at run-time inside the implementation of `JQuery` in JavaScript. However, this also applies to a class implemented in Scala.js code, which we can do as follows:

```
class JQueryImpl(element: HTMLElement)
    extends js.Object with JQuery {
  def html(): String = element.innerHTML
  def html(newValue: String): Unit =
    element.innerHTML = newValue
}
```

Since `JQueryImpl` is a JavaScript type, it has run-time dispatch semantics for overloaded methods. A call to `someJQueryImpl.html()` (from Scala.js or from JavaScript code) will resolve at run-time. Implementation-wise, the compiler generates the appropriate code to perform the overloading resolution at run-time.

The existence of the `js.Any` hierarchy and its distinct semantics is the core idea behind all of Scala.js' interoperability. We will explore all the details further in Section 4.

4. Interoperability Semantics

4.1 Type Correspondence

The attentive reader might have noticed that we glossed over an important detail in the example from Section 3. We have happily assumed that `String` accurately represents the values that jQuery expects and returns. If it doesn't, our previous code would be flawed, as it would call jQuery's `html()` method with something that the underlying JavaScript cannot understand.

Recall from Section 2 that GWT's interoperability specifies that a primitive `double` is seen by JavaScript as a primi-

[2] Although ES classes are often thought to be mere syntactic sugar over constructor functions and prototypes, it is not entirely true, as they have some unique semantics such as the inheritance restriction.

Scala types	Corresponding data type in JavaScript
`Boolean`	`boolean`
`Double`	`number`
`String`	`string` or `null` (`String` is nullable)
`Unit`	`undefined`
`Null`	`null`
`Byte`	Integer `number` in the range $[-2^7, 2^7 - 1]$
`Short`	Integer `number` in the range $[-2^{15}, 2^{15} - 1]$
`Int`	Integer `number` in the range $[-2^{31}, 2^{31} - 1]$
`Float`	`number` with 32-bit float precision

Table 1. Type correspondence in Scala.js

tive `number`, but that boxes thereof are not. In Scala.js, the language mandates the type correspondences shown in Table 1, regardless of whether those values enter generic contexts (or are upcast to `Any`). In other words, Scala.js does not exhibit the boxing issue. Implementation-wise, values of those types are never boxed in Scala.js. Instances of all other Scala types (including `Char` and `Long`) are specified as *opaque*. JavaScript sees them as objects with which it cannot interact. The system implemented in Scala.js allows to partially open up Scala types to JavaScript with *exports*, but a discussion of exports is omitted from this paper.

4.2 Manipulating Values of JavaScript Types

4.2.1 Method Calls

Recall from Section 3 that method calls on JavaScript types have run-time overloading dispatch. More importantly, they have JavaScript method call semantics. Intuitively, this means that a Scala.js method application of the form `x.meth(a, b)` where `x` is statically typed as a JavaScript type behaves as the JavaScript code `x.meth(a, b)`.

Past the intuition, however, this does not make sense, as `x`, `a` and `b` can be arbitrary Scala.js expressions, which are not valid JavaScript expressions in general. A better definition of the semantics of such a call would be: evaluate the Runtime Semantics of `x.meth(a, b)` according to the ECMAScript specification [4, §12.3.4], replacing invocations of `GetValue(x)` (resp. `a`, `b`) by the evaluation rules of the Scala language specification [16, §6] for `x` (resp. `a`, `b`). However, a completely formal derivation of the evaluation rules is out of the scope of this paper. We will therefore stick to the less formal definition of the semantics for the remaining of the paper, implying that evaluation of subexpressions is left to the Scala semantics, unless otherwise noted.

Result Values: Protecting our Borders Recall from Section 3 that the `html()` method is declared with an explicit result type of `String`. The Scala type system therefore expects and believes that calling `html()` will return a `String`. With JavaScript semantics, however, this might not be the case. The method could, for example, return an `Int`, if it is implemented in a native JavaScript class. If this happens, statically typed Scala code will start manipulating an `Int` value as if it were a `String`. This could have disastrous consequences. It is also extremely damaging for an optimizer, which cannot rely on a sound static type system to perform optimizations, even on Scala types.

To avoid these problems, we protect our borders by systematically checking that the values returned by JavaScript method calls conform to the static result type, in the fashion of [14, 22] (though without blame tracking). In other words, if `html()` returns an `Int`, a `ClassCastException` will be thrown. Explained as a code example, the call `x.html()` is actually equivalent to `x.html().asInstanceOf[String]`.

Because of type erasure in Scala, run-time type checks are always performed up to the erasure of a type, as defined by the Scala language specification [16, §3.7]. This means that a `list` of type `List[Int]` qualifies as `List[String]` for the purpose of this check, which therefore succeeds. It is only later, when accessing an element of that list, e.g., `list.head`, that an additional check will be performed, as `list.head.asInstanceOf[String]`. In Scala and in Scala.js, types are therefore only sound up to erasure. Even though erasure is often regarded as a liability, it gives Scala.js immunity against the common performance problems found in interoperability layers that check types at the borders [21]. Indeed, only the first-order type needs to be checked, which is a constant-time operation. Moreover, since typed function values already expose an untyped entry point due to erasure, Scala.js does not need wrappers for function values sent to dynamically typed parts, i.e., to JavaScript libraries, which means that function identity is not threatened.

In Scala.js, the expression `x.asInstanceOf[T]` is further specified as a no-op if the erasure of `T` is a JavaScript type. This also applies to manual `asInstanceOf` checks, as well as those automatically inserted for the erasure of generics. Consequently, *any* value can be successfully cast to any JavaScript type, making JavaScript types decidedly unsound (similarly to generic types). Operations applied to JavaScript types are always checked at run-time (by the JavaScript VM). This property makes JavaScript types behave similarly to the *like types* proposed by Wrigstad et al. [23], with the exception that casts from non-conforming types must be explicit.

Table 2 recaps all the interoperability semantics of expressions manipulating values of JavaScript types.

4.2.2 Function Call

Some JavaScript values are *callable*, i.e., they can be called with the function call notation `f(a1, ..., an)`. In Scala, a corresponding syntax exists, which desugars into calling the `apply` method of the function value, i.e., `f(a1, ..., an)` desugars into `f.apply(a1, ..., an)`. It is therefore natural to specialize the semantics of calling a method named `apply` into the semantics of a JavaScript function call. This allows to declare interfaces for JavaScript function values, similar to the ones for Scala function values. For example, a 1-argument function value can be typed as follows:

Scala.js declaration	Scala.js expression	Semantics as JavaScript code
Method calls (4.2.1)		
`def m(): R`	`x.m()`	⟦x⟧`.m()`
`def m(p1: T1,..., pn: TN): R`	`x.m(a1,..., an)`	⟦x⟧`.m(`⟦a1⟧`,...,` ⟦an⟧`)`
`def m(p1: T1,..., pi: Ti = _,..., pn: TN = _): R`	`x.m(a1,..., aj)`	⟦x⟧`.m(`⟦a1⟧`,...,` ⟦aj⟧`)`
`def m(p1: T1,..., pi: Ti, pn: TN*): R`	`x.m(a1,..., aj)`	⟦x⟧`.m(`⟦a1⟧`,...,` ⟦aj⟧`)`
`def m(p1: T1,..., pi: Ti, pn: TN*): R`	`x.m(a1,..., ai, an: _*)`	⟦x⟧`.m(`⟦a1⟧`,...,` ⟦ai⟧`, ...`⟦an.toJSArray⟧`)`
Function calls (4.2.2)		
`def apply(p1: T1,..., pn: TN): R`	`x.apply(a1,..., an)`	`(0,` ⟦x⟧`)(`⟦a1⟧`,...,` ⟦an⟧`)`
Property read (4.2.3)		
`val f: R`	`x.f`	⟦x⟧`.f`
`var f: R`		
`def f: R`		
Property write (4.2.3)		
`var f: T`	`x.f = v`	⟦x⟧`.f =` ⟦v⟧
`def f_=(v: T): Unit`		
Constructor function of a class (4.2.3)		
`class C extends js.Any`	`new C(a1,..., an)`	`new` ⟦js.constructorOf[C]⟧`(`⟦a1⟧`,...,` ⟦an⟧`)`
Native classes and objects (4.2.3)		
`@js.native class C`	`js.constructorOf[C]`	`<global>.C`
`@js.native object O`	`O`	`<global>.O`
Indexed properties and methods (4.2.3)		
`@JSBracketAccess def m(a: T): R`	`x.m(a)`	⟦x⟧`[`⟦a⟧`]`
`@JSBracketAccess def m(a: T1, b: T2): Unit`	`x.m(a, b)`	⟦x⟧`[`⟦a⟧`] =` ⟦b⟧
`@JSBracketCall def m(a1: T1, a2: T2,..., an: TN): R`	`x.m(a1, a2,..., an)`	⟦x⟧`[`⟦a1⟧`](`⟦a2⟧`,...,` ⟦an⟧`)`
Operators (4.2.3)		
`def unary_+ : R` (and `- ~ !`)	`+x`	`+`⟦x⟧
`def +(a: T): R` (and `- * / % << >> >>> & | ^ < > <= >= && ||`)	`x + a`	⟦x⟧ `+` ⟦a⟧
Instance tests (4.2.3)		
`class C extends js.Any`	`a.isInstanceOf[C]`	⟦a⟧ `instanceof` ⟦js.constructorOf[C]⟧
`trait T extends js.Any`	`a.isInstanceOf[T]`	compile error
`class C extends js.Any`	`a.asInstanceOf[C]`	⟦a⟧ (no-op)
`trait T extends js.Any`	`a.asInstanceOf[T]`	⟦a⟧ (no-op)

Table 2. Semantics of manipulating JavaScript types. `x` is assumed to have a static JavaScript type. For a result type `R`, results are cast to `R` as per `result.asInstanceOf[R]`. Each line of the table reads as: given a static declaration (in a JavaScript type) from the first column, the Scala.js expression from the second column has the same run-time semantics as the JavaScript code in the third column (where ⟦e⟧ denotes the evaluation of `e` according to the Scala specification.

```
trait Function1[-T1, +R] {
  def apply(a1: T1): R
}
```

so that a Scala call such as f(a1), which desugars into f.apply(a1), behaves as the JavaScript code f(a1).

4.2.3 Other Features

Due to space restrictions, we omit a detailed discussion of several additional interoperability features:

Property accesses: Both Scala and JavaScript have fields, getters and setters, obeying the Uniform Access Principle. The syntax for property access in Scala.js is mapped to the semantics of the corresponding syntax in JavaScript.

Loading native classes and objects: Top-level classes and singleton objects extending js.Any can be annotated with @js.native, indicating that they are not implemented in Scala.js, but rather in external JavaScript libraries.

Indexed properties and methods: JavaScript can dynamically access properties and methods with the bracket selection x[prop]. Since Scala does not have an equivalent syntax, these semantics are provided with the annotations @JSBracketAccess and @JSBracketCall in Scala.js.

Operators: Similarly to how methods named apply were mapped to JavaScript function calls in Section 4.2.2, methods whose name is one of the JavaScript symbolic operators are mapped to the semantics of the corresponding operator. Alphabetical operators such as typeof and instanceof are provided by primitive functions, e.g., js.typeOf(x) and x.isInstanceOf[C].

Together, these features can be used to accurately type the API of JavaScript arrays, for example:

```
@js.native class Array[A] extends js.Object {
  def length: Int = js.native
  def length_=(v: Int): Unit = js.native
  @JSBracketAccess
  def apply(idx: Int): A = js.native
  @JSBracketAccess
  def update(idx: Int, v: A): Unit = js.native
}
```

4.3 Creating JavaScript Values

Whereas Section 4.2 exhaustively showed how we can manipulate values of JavaScript types, this section highlights the features of Scala.js' interoperability that allow to create JavaScript values.

4.3.1 Values of Primitive Types

There are six primitive types in JavaScript: undefined, null, boolean, number, string and symbol. With the exception of symbol, all of them have equivalent Scala types as defined in Section 4.1. The regular Scala constructs to create values of those types (such as literals) can be used to create the JavaScript values, as they are the same. Symbols are created as in JavaScript, using the functions Symbol(desc) and Symbol.for(key), which can be called through the interoperability semantics for function call and method call.

4.3.2 Function Values

Even though Scala has syntax to create function values, they are Scala function values, which are not equivalent to JavaScript function values. To convert a Scala function into a JavaScript function, we provide primitive functions js.Any.fromFunction0 through fromFunction22. For example, for functions of one argument, we have:

```
def fromFunction1[T1, R](
    f: T1 => R): js.Function1[T1, R]
```

Recall from Section 4.2.2 that js.Function2 is a type describing a JavaScript callable with 2 arguments. Such a conversion function can be used as follows:

```
val g = js.Any.fromFunction1((a: Int) => a+1)
```

Formally, fromFunction1 returns a new function object, as defined in [4, §9.3], that, when called, forwards the call and arguments to the apply method of f. In other words, the JavaScript code g(a, b) has the Scala semantics of f.apply(a, b). Note that such function values always discard the value of thisArgument that they are given, as do arrow functions in JavaScript. An additional series of js.ThisFunction types explicitly capture the thisArgument as an additional formal parameter at the Scala.js level.

Using the methods fromFunctionN is quite cumbersome. The practical system implemented in Scala.js uses Scala's implicit conversions to remove the extra verbosity.

4.3.3 Class Values

In ECMAScript 5.1 and earlier, class values were no different than function values combined with prototype inheritance, which can be created with the interoperability features we have already seen. However, in ECMAScript 2015, class values are a distinguished feature with some specific semantics, for which we need dedicated support.

Since most parts of class definitions straightforwardly to corresponding concepts in JavaScript classes, we omit a detailed discussion, and summarize the semantic equivalences in Table 3. Note that formal parameters are checked with casts to protect the borders, since JavaScript code calling the method could give arbitrary parameters.

One aspect deserves to be further discussed, though, namely overloading. In Section 3, we saw that overloading in JavaScript types has run-time dispatch semantics. Section 4.2.1 provided precise semantics for method calls, but we now need to give semantics to method definitions. Consider the following definitions:

```
def add(x: Double, y: Double): Point =
  new Point(this.x + x, this.y + y)
def add(that: Point): Point =
  new Point(this.x + that.x, this.y + that.y)
```

Scala.js definition	Semantics as JavaScript code
`class C extends D with ...`	`class C extends ⟦js.constructorOf[D]⟧`

`val f: T`
`var f: T`
	`this.f = ⟦defaultOf[T]⟧`, in the constructor

`def m(p1: T1, ..., pn: TN): R =`
`  expr`
	`m(q1, ..., qn) {`
	`  const p1 = ⟦q1.asInstanceOf[T1]⟧; ...;`
	`  return ⟦expr⟧;`
	`}`

`def f: T = expr`
`def f_=(v: T): Unit = stat`
	`get f() { return ⟦expr⟧; }`
	`set f(w) { const v = ⟦w.asInstanceOf[T]⟧; ⟦stat⟧ }`

`class C(p1: T1, ..., pn: TN)`
`    extends D(a1, ..., am) {`
`  stats`
`}`
	`constructor(q1, ..., qn) {`
	`  const p1 = ⟦q1.asInstanceOf[T1]⟧; ...;`
	`  super(⟦a1⟧, ..., ⟦am⟧);`
	`  ⟦stats⟧;`
	`}`

Table 3. Semantics of the definition of JavaScript classes (without overloading considerations).

In JavaScript class definitions, there can only be one method add, which should handle both cases. That method will perform run-time dynamic dispatch to the appropriate overload. The dispatch uses a combination of testing the number and run-time types of the actual arguments. For the above case, we can express the run-time tests as follows (abusing the Scala syntax `.asInstanceOf[T]` within JavaScript):

```
add(...args) {
  switch (args.length) {
    case 1:
      return this.add_Point(
          args[0].asInstanceOf[Point]);
    case 2:
      return this.add_Double_Double(
          args[0].asInstanceOf[Double],
          args[1].asInstanceOf[Double]);
    default:
      throw new TypeError(
          "No matching overload");
  }
}
```

The general algorithm for run-time dispatch is as follows:

1. Switch on the number of actual arguments for each group of overloaded definitions with the same number of formal parameters.
2. For each value of the number of formal parameter count:
   (a) If there is only one overloaded definition, call it, with appropriate type checks for the actual arguments.
   (b) Otherwise, find the first parameter position at which the type is not the same in all definitions. Perform run-time type tests on that parameter to refine the set of possible definitions, and go back to step 2a.
   (c) If all the erased types are equal, throw an error. Note that the existence of this case can be decided at compile-time, and is reported as a compile error.

The order in which run-time type tests are performed in step 2b matters, as there are subtyping relationships between some types. In this case, most specific types are tested first.

Default parameters and variadic parameters are handled in a similar way.

The feature interaction between overloading and inherited methods is even more challenging. Consider the following Scala.js classes:

```
class Parent extends js.Object {
  def foo(x: Double): Double = x * 2
}
class Child extends Parent {
  def foo(x: String): String = x + "hello"
}
```

What should the semantics of `Child.foo` be? If it only handles the `String` case, then `Parent.foo` fails to be inherited, as it would be shadowed by the redefinition of `foo` in `Child`. `Child.foo` should therefore dispatch among all the alternatives of `foo`, including those inherited from its parent classes. In the cases where the dispatch resolves to an inherited method, it should delegate to the parent implementation (which, in turn, might have to re-do an overloading dispatch).

```
class Child extends Parent {
  foo(arg1) {
    if (arg1.isInstanceOf[Double])
      return super.foo(arg1);
    else if (arg1.isInstanceOf[String])
      return arg1 + "hello";
    else
      throw new TypeError(
          "No matching overload");
  }
}
```

4.4 Completeness

Now that we have gone through all the interoperability features of Scala.js, it is time to revisit our initial goad.

Recall from Section 2 that, in order to be able to talk to any JavaScript libraries, we need our interoperability features to cover "all of JavaScript". We argue that our interoperability features are *complete*, in the sense that they support "all of JavaScript" (in Strict Mode [4, §10.2.1]). But what exactly *is* "all of JavaScript"? We find the answer in the ECMAScript specification [4], sections 10 to 15, included, which are entitled "ECMAScript Language". Supporting "all of JavaScript" essentially means being able to express all of the run-time semantics offered by the ECMAScript language.

Table 4 shows a comprehensive list of the relevant subsections in the ECMAScript specification, together with links to the interoperability features described in this paper which give access to them. Note that the ECMAScript specification defines both static and run-time semantics. The former being more about JavaScript source code than evaluation semantics, only the latter are relevant. In particular, we entirely skip Sections 10 and 11 entitled "Source Code" and "Lexical Syntax", respectively. We also skip language constructs which are obviously replaced by corresponding Scala language constructs, such as identifiers and control structures.

As we can see, there are exactly four features that are not implemented: `new.target`, generator functions, static methods and modules.

Generator functions are syntax sugar over normal functions. If needed, they can be implemented using other constructs, although there will be some syntax overhead. It is therefore still *possible* to achieve their semantics in Scala.js, even though it might not be convenient. In the future, the syntax overhead could be removed using a Scala macro, which could perform the same desugaring in terms of Scala.js constructs.

Static methods are essentially function properties of the class value. Scala does not have static methods, which makes it very difficult to provide a declarative syntax for ECMAScript static methods. The limitation can be worked around by attaching function values to properties of the class value using imperative statements, such as

```
js.constructorOf[C].increment =
  (a: Int) => a + 1
```

There are plans to add support for static methods in Scala [17], which will allow Scala.js to have better support in the future.

`new.target` and imports are truly missing pieces of semantics that cannot be otherwise represented. They will be the topic of future work, as well as static methods. We are confident that the methodology we have applied so far can be extended to the last unsupported bits.

5. Related Work

Language interoperability is an important problem that has been explored both in the theoretical literature as well as in concrete implementations.

Matthews and Findler [14] develop a theoretical foundation for interoperability between a statically typed language and a dynamically typed language. They define a Natural Embedding that allows to embed Scheme-like programs into ML-like programs and vice versa. At the boundaries, numbers and functions are converted on a type-directed basis between the Scheme representations and ML representations, with dynamic type tests to ensure that values match their types when flowing from Scheme to ML. Function values are only checked up to their first-order behavior, delaying further checks for the arguments (or result values) to the application of the functions. We do something very similar in Scala.js: values coming from JavaScript semantics and flowing into statically typed Scala code are downcast to their erasure, which is essentially equivalent to Matthews and Findler's first-order behavior. Dynamic checking of static contracts is also a recurring theme in gradual typing [20, 22], although the run-time semantic difference issue does not apply in those cases. Gradually typed systems often suffer from performance problems [21] due to either deep type tests or towers of wrappers. Scala.js avoids those issues by blending type tests related to interoperability borders with those necessary due to erased generic types in Scala.

Our interoperability features go beyond converting data representations at language boundaries, however. Besides numbers and functions, Scala.js can manipulate arbitrary JavaScript objects, including mutable values. For those, converting at language boundaries is not an option, as mutable operations would not be carried over. We achieve this deeper level of interoperability by unifying Scala values and JavaScript values in one type system, capable of representing both. To the best of our knowledge, there has been no formal study of such a system yet, in the context of interoperability between languages with different run-time semantics. In particular, the interaction between compile-time- and run-time overloading resolution seems unaddressed, which would constitute interesting future work.

Type-based representations of foreign language mutable objects has been the topic of several practical systems, however. Some early work was done by Elsman in SMLtoJs [6], using phantom types to represent JavaScript values. That system considered foreign values as blackboxes that could not be directly manipulated by SML code. More recent developments include the overlay types and JsTypes of GWT [11], which do allow direct manipulation. However, as we saw in Section 2, their semantics have several limitations due to their implementation-based behavior.

Independently of the run-time semantics problem, typing native APIs with Scala.js traits, classes and objects was initially heavily inspired by TypeScript's type definitions [15]. We adapted some language features to fit Scala's type system, such as using `@JSBracketAccess` annotations where TypeScript has dedicated syntax. We later expanded the framework to support the definition of class values from Section 4.3.3.

ECMAScript 2015 language features		Scala.js interoperability features	
12.2.2	`this`	4.3.2	`this` in methods; `js.ThisFunction` in functions
12.2.4	literals	4.1	Scala literals and type correspondence
12.2.5	array initializer	4.2.2	`js.Array(e1, ..., en)`, implemented with function call
12.2.6	object initializer	4.3.3	`new js.Object { val f1 = v1; ... }`, an anonymous class
12.2.8	regex literals	4.2.3	can be implemented with `lazy val re = new js.RegExp("re", "flags")`
12.3.2	property accessors	4.2.3, 4.2.3	property access and indexed properties
12.3.3	the `new` operator	4.2.3	`new` for JavaScript classes
12.3.4	function calls	4.2.1, 4.2.2	method calls and function calls
12.3.5	the `super` keyword	4.3.3	`super` references
12.3.8.1	`new.target`		*not supported*
12.4.{4–5}, 12.5.{7–8}	`++` and `--`		can be implemented with `+= 1` and `-= 1`
12.{4–12}	other operators	4.2.3, 4.2.3	JavaScript operators and instance tests
13.7.5	`for-in` and `for-of` loops		implemented as user code in the Scala.js standard library
13.16	the `debugger` statement	4.2.3	primitive `js.debugger()`
14.{1–2}	(arrow) function definition	4.3.2	conversions from Scala function values
14.3	method definitions	4.3.3, 4.3.3	methods, getters and setters in JavaScript classes
14.3	static methods		*not directly supported*, can be assigned after the class has been created
14.4	generator function definitions		*not directly supported*, but they desugar into other concepts that are supported
14.5	class definitions	4.3.3	class value definitions
15.2	modules		*not supported*

Table 4. ECMAScript 2015 language features and the corresponding interoperability features of Scala.js. Each line of the table reads as: the run-time semantics of the JavaScript construct from the first column (as specified in the referenced section of [4]) can be expressed in Scala.js using the interoperability feature from the second column (as specified in the referenced section of the present paper).

6. Conclusion

We have shown in Sections 3 and 4 how we can extend Scala's type system with an additional type hierarchy for JavaScript types. Operations on values of these types have different run-time semantics than traditional Scala values, namely ECMAScript semantics. This change of semantics closes the impedance mismatch between Scala and JavaScript, allowing to talk to JavaScript APIs from Scala.js code. We have argued in Section 4.4 that our interoperability features are virtually complete with respect to the ECMAScript 2015 language: they offer all the run-time semantics that are offered by ECMAScript. Two missing features, namely the meta-property `new.target` and modules, are yet to be covered, but we are confident that we can address them using a similar approach. We say that this approach is semantics-driven because it provides language constructs in Scala.js that have the semantics of ECMAScript constructs, so that "all of ECMAScript" is supported in Scala.js. To the best of our knowledge, Scala.js is the first system providing "all of ECMAScript", by that definition.

We believe that the same approach could be used by a variety of other statically typed language, including GWT, at least if their paradigms are not too different from the dual object-oriented/functional nature of JavaScript: identify a particular set of types dedicated to interoperability (in Scala.js, subtypes of `js.Any`), then specify a set of semantics for operations on these types to cover JavaScript's semantics. The particular combination of Scala and JavaScript gives a pleasant syntax, because the two languages are syntactically close, but it need not be so. We have seen examples even in Scala.js where the syntax is not identical in both languages, such as for indexed properties in Section 4.2.3.

All the interoperability semantics presented in this paper have been implemented in the standard distribution of Scala.js [19]. They have been used to write type definitions for a large number of JavaScript libraries, including jQuery, React and Angular[3], which allows all these libraries to be called from Scala.js.

Acknowledgements This work was partially supported by the Swiss Commission for Technology and Innovation (CTI) grant 16613.1.

References

[1] ClojureScript. `http://clojure.org/clojurescript`, 2015.

[2] CoffeeScript. `http://coffeescript.org/`, 2015.

[3] S. Doeraene. Scala.js: Type-directed interoperability with dynamically typed languages. Technical Report EPFL-REPORT-190834, École polytechnique fédérale de Lausanne, Switzerland, 2013.

[4] Ecma International. *ECMAScript 2015 Language Specification*. Geneva, 6th edition, June 2015. URL `http://www.ecma-international.org/ecma-262/6.0/`.

[5] Elm. `http://elm-lang.org/`, 2015.

[6] M. Elsman. SMLtoJs: Hosting a standard ML compiler in a web browser. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*, PLASTIC '11, pages 39–48, 2011.

[7] Facebook. Flow. `http://flowtype.org/`, 2015.

[8] H. Foundation. Haxe. `http://haxe.org/`, 2015.

[9] FunScript. `http://funscript.info/`, The F# Software Foundation.

[10] Google. Dart. `https://www.dartlang.org/`, 2015.

[11] Google. Google Web Toolkit. `http://www.gwtproject.org/`, 2015.

[12] Google. JsInterop 1.0, nextgen GWT/JavaScript interoperability. `http://bit.ly/1IrvE2M`, 2015.

[13] P. S. Ltd. Phaser. `http://phaser.io/`, 2015.

[14] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31:12:1–12:44.

[15] Microsoft. TypeScript. `http://www.typescriptlang.org/`, 2015.

[16] M. Odersky. *Scala Language Specification*. 2.11 edition, 2013.

[17] D. Petrashko, S. Doeraene, and M. Odersky. `@static` fields and methods in Scala objects – Scala Improvement Proposal. `https://github.com/scala/scala.github.com/pull/491`, 2016.

[18] PureScript. `http://www.purescript.org/`, 2015.

[19] Scala.js. `http://www.scala-js.org/`, 2015.

[20] J. Siek and W. Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming*, ECOOP '07, pages 2–27, 2007.

[21] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 456–468, 2016.

[22] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 964–974, 2006.

[23] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 377–388, 2010.

[3] A list can be found at `https://www.scala-js.org/libraries/facades.html`.