Scala.js: Type-Directed Interoperability with Dynamically Typed Languages

Sébastien Doeraene

École polytechnique fédérale de Lausanne
sebastien.doeraene@epfl.ch

Abstract

Interoperability between statically typed and dynamically typed languages is increasingly important, as can be witnessed by the many statically typed languages targeting JavaScript. Interoperating with both the object-oriented and functional features of JavaScript is essential, if only to manipulate the DOM, yet existing languages have very poor support for this.

We present Scala.js, a dialect of Scala compiling to JavaScript. Its interoperability system is based on a powerful and intuitive framework for type-directed interoperability with dynamically typed languages. The framework combines facade types for JavaScript values; user-defined, implicit, type-directed cross-language conversions; and a Dynamic type building on facade types and implicit conversions. It accommodates both the functional and object-oriented features of Scala and JavaScript, and provides very natural interoperability between the two languages. It is expressive enough to represent the DOM and jQuery APIs, among others, both in its statically typed and dynamically typed flavors.

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

General Terms Languages, Design

Keywords interoperability, static types, dynamic types, Scala, JavaScript

1. Introduction

With the advent of JavaScript as a platform for large-scale Web application development rather than tiny scripts on Web pages, a number of efforts have been made to compile higher-level languages down to JavaScript. In particular, source languages supporting some form of static typing are common.

We can divide these languages in two radically different approaches. The first approach is to start from JavaScript itself, and either provide new syntactic sugar (e.g., CoffeeScript [7]), or add various forms of static typing. Typing JavaScript, as any other dynamically typed language, is hard. Various strategies have been studied both for JavaScript itself, or for idioms found in JavaScript: soft typing [4, 26], gradual typing [15, 24, 25], dependent types [5], or occurrence typing [27]. However, none of these approaches for types has really made it for JavaScript, the aborted ECMAScript 4 proposal being the most prominent example of failed attempt. An apparently successful approach is conducted by TypeScript [18], which has simply chosen to embrace unsoundness. The golden rule or motto of all these languages is some variation of “It is just JavaScript.”

Although interesting and useful, these approaches do not add that much expressiveness to the language. They do not add language constructs or concepts that cannot be translated trivially and locally into JavaScript.

The other approach is to start from a very different language, designed with application-scale development in mind, typically involving static typing. Both implementations of new languages designed specifically for JavaScript (Dart, Haxe, etc.) and ports of existing languages (Java, Standard ML, etc.) have been done. This approach typically gives more powerful languages, with higher abstractions, and that also avoid the typical quirks of JavaScript (lack of lexical scoping, non-trivial semantics for operators, etc.). However, all of these languages perform poorly in terms of interoperability with JavaScript libraries.

Now, interoperability with JavaScript is *essential*, if only because this is the way to interact with the Document Object Model (DOM). Languages taking this approach define their own API to interact with the DOM, and provide obscure bindings under the hood. But this is not very flexible, as it does not scale well to user-defined JavaScript libraries, often requiring the developer to use embedded JavaScript from within the host language. It is also difficult to call such code from JavaScript. The reason for this poor interoperability is that these languages do not consider JavaScript values and objects to be first-class citizens of the host language. Additionally, with the exception of Dart (which is only partially typed), their commitment to a static type system prevents the developer to interact with JavaScript libraries without giving proper types of some sort to them.

Scala.js. In this paper, we present Scala.js, yet another language compiling to JavaScript. As its name implies, it is a port of the popular functional and object-oriented language Scala [20]. Scala has a very powerful type system with a unique combination of features: traits, implicit conversions, higher-order functions, generics, and user-defined dynamic types. As a functional and object-oriented language, its concepts are also very close to JavaScript, behind the type system: no static methods (only objects), first-class functions, etc.

Scala.js is obviously part of the second set of languages targeting JavaScript. However, it was designed with interoperability in mind, and as a result provides the best interoperability with JavaScript of all these languages that we know of. Ironically, Scala.js leverages the powerful type system of Scala to provide interoperability with the dynamically typed language JavaScript. In particular, it is built on two unusual features of the Scala type system: user-defined implicit conversions and user-defined dynamic types.

Type-directed interoperability between statically typed and dynamically typed languages is not novel, but they have mostly covered only functional languages [17]. Most importantly, they only allow builtin conversions from JavaScript values from and to the host language data types, i.e., there is no user-defined conversions. Another limitation is that they either (a) do not support untyped JS values at all or (b) consider them as blackboxes.

Scala.js combines the best of existing state-of-the-art frameworks (type-directed interoperability [17], static type facades for object-oriented JS types [18], the Scala type system itself) with novel ideas of its own (user-defined cross-language conversions, type-directed interoperability in the presence of object-orientation and a type-directed dynamic type) to provide outstanding interoperability with JavaScript.

Scala.js has been successfully implemented as a new backend for the Scala compiler, that is loaded as a compiler plugin. It supports the whole Scala language, modulo some semantic adaptations (most notably for overflows in arithmetics), and programs written in Scala.js can use the Scala collection library, as well as interoperate with JavaScript libraries. The interoperability framework is built on top of implicit conversions and user-defined dynamic types of Scala, so that the type system need not be modified. The compiler supports separate compilation and incremental compilation (just as the standard Scala compiler does).

Contributions. This paper brings the following contributions.

We build upon existing type-directed interoperability for functional features [17] and show how to extend it to mutable object-oriented values through first-class static type facades, and how to allow for user-defined, type-directed cross-language conversions.

We also show how to integrate dynamically typed values of JavaScript as first-class citizens of a statically typed language, and how type-directed interoperability still applies in that setting, providing whitebox, intuitive interoperability.

Finally, we show how the framework can be built on top of user-defined implicit conversions, and user-defined dynamic types, as available in Scala, with no changes to the core language and its type system.

2. Overview of Scala.js

Scala.js is a dialect of Scala which is meant to be compiled to JavaScript, instead of the JVM. The goal of this project is to allow front-end developers to write the client-side of Web applications entirely in Scala, and have it compiled to standard JavaScript [8] for execution in Web browsers, as part of Web pages.

Although described as a dialect, Scala.js supports all the type system and all the constructs of Scala, including pattern matching, mixin composition, and so on. As such, Scala.js can compile any existing Scala code. Actually, Scala.js compiles the entire Scala standard library (modulo system-dependent parts) which is therefore available to Scala.js programs and libraries. It is a dialect because the semantics of some primitive operations are a bit different, mostly to accommodate the fact that only `Doubles` exist in JavaScript.

Moreover, Scala.js features outstanding interoperability with JavaScript libraries. Unlike previous similar attempts, which pride themselves on supporting some limited, simple, and sometimes awkward interactions with the DOM [9], Scala.js supports *all* interactions with the DOM and any other user-defined library, within a simple and consistent framework for type-directed interoperability.

Support for all of Scala. The Scala.js compiler supports all of the Scala language. Hence, it can compile the entire Scala standard library, which is then available to programs in Scala.js. Hence the following program, which is valid Scala, is also valid Scala.js and will print the first 10 squares to the standard output (in a browser, the "standard output" is the logging buffer, accessible through the `console.log()` function):

```
object Main {
  def main(): Unit = {
    val nums = (1 to 10).toList
    val squares = nums map (x => x*x)
    for ((x, x2) <- nums zip squares)
      println(s"$x -> $x2")
  }
}
```

Scala.js emits standard JavaScript, which can be run by any JavaScript interpreter conforming to ECMAScript 5.1 (in browsers, or standalone such as d8 from V8 or Node.js, or embedded such as Rhino). The Scala.js development kit (taking the form of an sbt plugin) includes a builtin mechanism to run Scala.js programs through Rhino, for testing purposes (including unit testing).

Since Scala as a language, and also its standard library, rely on core parts of the Java standard library, it is impossible to support all of Scala without supporting some of Java. Hence, Scala.js also includes a partial port of the Java standard library written in Scala.js itself.

Finally, it is also worth mentioning that Scala.js does support macros [3] and compiler plugins [19], just as regular Scala; although this is not the result of our work, but rather due to the modularity of the Scala compiler.

Semantics for numbers. Scala has defined its primitive types, and in particular, its numeric types, with the JVM in mind, which offers signed integers of sizes 8, 16, 32 and 64 bits, and floating point numbers of size 32 and 64 bits. JavaScript, however, has only one number type, equivalent to the `Double` type of Scala. Although it would be possible to encode faithfully the wrapping semantics of integers of various sizes, the runtime overhead would be too great. Every primitive arithmetic operation would have to be programmed with non-trivial JavaScript code.

Our implementation instead translates all numeric types (as well as characters) to JavaScript numbers. The semantics are preserved for integers as long as no logical overflow happens, i.e., an operation that would overflow in Scala, but does not in JavaScript. Integer division is preserved by truncating the result of the division between two numbers statically typed as integers.

This approach for encoding numbers has been used by all major ports of existing languages to JavaScript, e.g., GWT [13] and ClojureScript [6].

Runtime reflection. Java and, by extension, Scala, have a relatively powerful runtime reflection subsystem. Scala.js, given considerations about aggressive code optimization (see next paragraph), does not support runtime reflection. Limited support is given to support heavily used idioms of Scala. This includes (a) runtime type tests (`isInstanceOf[T]`), including for interfaces and (b) `ClassTags`. Full-fledged reflection could be implemented without much trouble, and is intended for future work.

Fast development cycle and optimized production code. The Scala standard library is huge, by JavaScript's standards. To support fast, incremental development cycles, which is dear to front-end developers, Scala.js supports separate compilation, and even incremental compilation. It also emits source maps, which allows to debug the original Scala.js code instead of the obscure, emitted JavaScript code.

For production deployment, a much smaller code is necessary. Scala.js produces code that can be optimized by the Advanced Optimizations mode of Google's Closure Compiler [11]. These optimizations use whole program analyses to perform aggressive renaming, dead code elimination, namespace flattening, and so on.

Interoperability with JavaScript. All of the above would be worthless if Scala.js was not able to interoperate with JavaScript code, as we explained in the introduction. Scala.js features a unique, type-directed interoperability, which is the main contribution of our work, and which we describe in details in Section 4.

3. Fancy features of the Scala type system

Before describing our interoperability framework, we give an introduction to the two "fancy" features of the Scala type system on which we build the framework. They are user-defined, type-directed implicit conversions, and user-defined dynamic types. We will show that our framework can be implemented without changing the host language's type system, as long as it supports these two features.

3.1 Implicit conversions

In Scala, one can define implicit conversions as methods with the `implicit` keyword, e.g.,

```
case class ID(val id: String)
implicit def stringToID(s: String): ID = ID(s)
```

When an implicit conversion from some type `A` to some other type `B` is in the lexical scope, and a term is typed as an `A` but is expected to be of type `B` (or some supertype of `B`), the Scala type-checker inserts a call to the implicit conversion. For example, given the implicit conversion from `String` to `ID` defined above, the following code

```
def lookup(id: ID): Book = { ... }
val book = lookup("foo")
val id: ID = "bar"
```

is valid, because the type-checker will rewrite it as

```
val book = lookup(stringToID("foo"))
val id: ID = stringToID("bar")
```

Implicit conversions can also be invoked automatically when selecting a member `m` of some value of type `A`, if `m` is not defined in `A`, but there exists an implicit conversion from `A` to some other type `B` in scope, and `m` is defined in `B`. This is typically used for defining extension methods, as illustrated by the following snippet, extracted from the Scala standard library:

```
class RichInt(self: Int) {
  def to(end: Int): Range =
    Range.inclusive(self, end)
}
implicit def int2richInt(i: Int): RichInt =
  new RichInt(i)
```

which is why we can write

```
val nums = (1 to 10).toList
```

which is equivalent to

```
val nums = 1.to(10).toList
```

which itself is rewritten by the type-checker into

```
val nums = int2richInt(1).to(10).toList
```

Implicit values in scope can also be instantiated implicitly to be given to implicit parameters with the appropriate type, as studied in [21, 22]. Since our framework does not use this aspect of implicits directly, we do not include the details here.

3.2 User-defined dynamic types

Since version 2.10, Scala features a special trait, `scala.Dynamic`, which can be used to define custom dynamic types. This trait itself does not define any member, but can be mixed into a user-defined class or trait to enable the dynamic rewriting upon access of a member of this type. When selecting a member `x.m` of a value `x` of some type `D` $<:$ `scala.Dynamic`, and `m` cannot be found by all other type-checking rules (direct selection, but also selection available through an implicit conversion in scope), the following rules apply.

- If `m` is selected as read access of a field, i.e., under the form `x.m` anywhere but in the left-hand side of an assignment, the type-checker rewrites `x.m` as `x.selectDynamic("m")`.
- If `m` is selected as write access of a field, i.e., under the form `x.m = y` for some expression `y`, then the type-checker rewrites `x.m = y` as `x.updateDynamic("m")(y)`.
- If `m` is selected as a method call, i.e., under the form `x.m(arg1, ..., argn)`, then the type-checker rewrites the call as `x.applyDynamic("m")(arg1, ..., argn)`.[1]

The rewritings are type-checked recursively, with the exception that the dynamic rewriting cannot be used for `x` anymore (it can be used when type-checking subexpressions).

For example, given the following definition:

```
class DynExample extends scala.Dynamic {
  def applyDynamic(n: String)(args: Int*): Int = ...
  def selectDynamic(n: String): Int = ...
  def updateDynamic(n: String)(v: Int) = ...
}
```

the following snippet is valid

```
val d = new DynExample
val i: Int = d.foo(5, 3)
d.bar = d.foobar * 2
```

since it can be rewritten as

```
val i: Int = d.applyDynamic("foo")(5, 3)
d.updateDynamic("bar")(
    d.selectDynamic("foobar") * 2)
```

4. Type-directed interoperability with JavaScript

In addition to being able to support the entire Scala language, a crucial design requirement of Scala.js was to have outstanding, easy-to-use and intuitive interoperability with JavaScript code. This section presents type-directed interoperability between Scala.js and JavaScript code, which is the main contribution of this paper. Using a combination of facade types, type-directed user-defined conversions, and a `Dynamic` type (itself also type-directed), Scala.js features the most powerful type-directed interoperability with a dynamically typed language that we know of.

Although we present the features in the context of Scala.js, they could be applied to any such language, as long as the type system of the host language supports the appropriate features, i.e., user-defined dynamic types and implicit conversions. Ideally, both languages should support the same main paradigms, which is the case for Scala and JavaScript which both combine object-oriented and functional features.

4.1 Why interoperability matters

Before discussing type-directed interoperability, it is worth understanding why interoperability matters at all. Why cannot we just do *nothing*? After all, ours is not an interoperability between two different runtime systems (like JNI which is JVM-C). The interoperability discussed here is one between *languages*. But why is it hard?

Supporting all of Scala is great, but it has consequences. Two representative features that require particular care are overloading, and `isInstanceOf` tests (necessary to support pattern matching). JavaScript has neither of those, although one can argue that JavaScript supports callee-handled overloading using type tests and testing the number of arguments passed to the function; and that there exists the `instanceof` operator. However, caller-handled overloading cannot reproduce all the behavior of compiler-handled overloading, because it does not have the static type of the arguments (it only has their dynamic type). And `instanceof` is only meaningful for *classes*, not for interfaces or traits, since interfaces

[1] A fourth rule exists for named parameters, which we do not used here.

do not even exist in JavaScript. In order to support those features (and some others), Scala.js encodes the classes and traits it emits.

Overloading is simple to encode, starting from the specifications of both languages. In JavaScript, the choice of the method to apply is only dictated by its *name*; whereas in Scala (or more precisely, in the JVM), the method is identified by its name, the erased types of its formal parameters and its erased result type. The logical conclusion is to encode the JavaScript name so that it contains all of the JVM identity: the name of the method, and the fully qualified names of the parameter types and result type. For example, the following functions

```
def foo(x: Int): Int
def foo(x: String): Int
def bar(x: Any, y: Int => Int): Unit
```

would have the following names in JavaScript, respectively,

```
foo__I__I
foo__Ljava_lang_String__I
bar__Ljava_lang_Object__Lscala_Function1__V
```

where `I` stands for the primitive `Int` and `V` for `Void` (i.e., `Unit`).

Supporting accurate `isInstanceOf` tests is more involved. Scala.js adds a special field `$classData` to the prototype of the classes it emits. Its value is an instance of a supporting JavaScript class `ClassTypeData`, containing various runtime type information of the class. Among others, it contains a field `ancestors`, which is a dictionary whose keys are the fully qualified name of all the ancestors of the class (including itself, superclasses, and also interfaces it implements), and whose values are all `true`. For example, the dictionary for the class `scala.Option` is

```
{
  scala_Option: true,
  scala_Serializable: true,
  java_io_Serializable: true,
  scala_Product: true,
  scala_Equals: true,
  java_lang_Object: true
}
```

Given this structure, we can implement, e.g., the instance test for the `Product` interface like so:

```
function isInstanceOf__scala_Product(obj) {
  return !!(obj && obj.$classData &&
      obj.$classData.ancestors.scala_Product);
};
```

Other encodings are possible, and deciding which one to use is a matter of optimization, which will be investigated in future work. The point we want to make here is that, in order to support all of Scala, we cannot just translate Scala classes to bare JavaScript classes. It is necessary to apply some sort of encoding that produces classes in JavaScript that do not look like JavaScript classes. For the same reason, we cannot implement Scala arrays as bare JavaScript arrays; nor can we implement Scala functions as bare JavaScript functions.

At a more fundamental level, and this is why the discussion is relevant here, it means that Scala values and corresponding JavaScript values cannot always have the same runtime data representation: Scala.js classes have an additional field, and their method names are mangled. Hence, the need for some sort of interoperability machinery.

New languages targeting specifically JavaScript completely avoid that kind of concerns by not introducing such constructs in the first place. If Scala.js wanted to avoid these issues, then it would not be Scala anymore. There would be no pattern matching, no overloading, no functions-are-just-classes property, and so on.

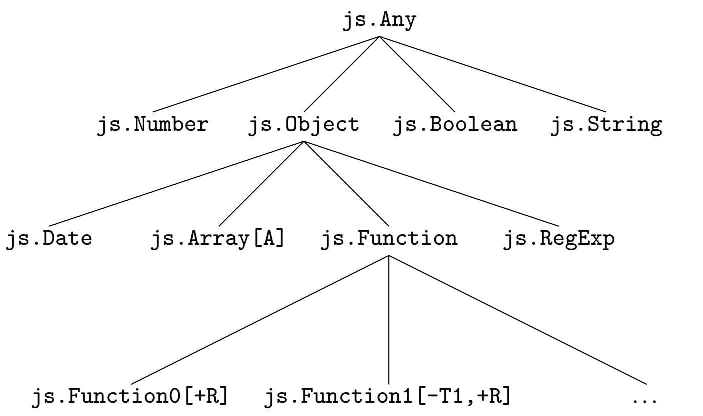

Figure 1. Core JS types in Scala.js

4.2 Facade types

The interoperability with JavaScript is entirely type-driven. Figure 1 shows the core types defined in the Scala.js standard library, which are facade types for standard ES5 data types. The various `js.Function`N types represent functions of various arities, just like `Function`N types in Scala do. They are obviously contravariant in parameter types (`-T1`) and covariant in result types (`+R`).

Facade types are first-class citizens of the host language's type system: they can participate in higher-order types, have type parameters, have abstract type members, and so on. For example, one can have a `List[js.Number]` or a `js.Array[Option[Int]]`. Similarly, values of facade types are first-class terms in the host language: they can be stored in variables of the appropriate type, can be properly assigned to values of type `Any`, their methods can be called using regular syntax, and so on. This is in contrast to previous efforts at interoperability, in which such values are considered as blackboxes, and the types are at best phantom types in the host language (e.g., in SMLtoJs [9]). In both cases, they just do not feel right or natural at all. Our facade types, being first-class citizens of the host type system, feel very natural and easy to use.

The compiler knows that `js.Any` and its subtypes are facade types for JavaScript values. It does not emit code for these. And when calling a method of such a type, e.g., `regexp.exec(s)`, the compiler translates these as so-called raw JS method calls. Basically, this disables Scala-related encodings, such as overload mangling, accessors for `val`s and `var`s, and so on.

It is possible to declare, in user code, facade types for other JavaScript types, by declaring traits inheriting directly or indirectly from `js.Object`. A typical example would be types for the DOM API:

```
trait Window extends js.Object {
  def alert(msg: js.String): Unit
}

object Main {
  def main(win: Window): Unit = {
    win.alert("Hello world!")
  }
}
```

Constructible types, i.e., whose instances can be constructed with `new` in JavaScript, can simply be declared as facade classes. Moreover, top-level objects in JavaScript can be given a facade with Scala top-level objects inheriting from `js.Object`. A special marker trait, `js.GlobalScope`, can be used to mark top-level objects (including package objects) that give facade types to top-level values, variables and functions, i.e., they represent the top-level, global scope of JavaScript. The following code uses such facade types to manipulate the DOM.

```
object Window extends js.GlobalScope {
  val document: DOMDocument
}
trait DOMDocument extends js.Object {
  def getElementById(id: js.String): HTMLElement
}
trait HTMLElement extends js.Object {
  def appendChild(child: HTMLElement): Unit
}
class Image extends HTMLElement {
  var src: js.String
}

object Main {
  def main(): Unit = {
    val playground =
      Window.document.getElementById("playground")
    val img = new Image
    img.src = "./path/to/img.png"
    playground.appendChild(img)
  }
}
```

The compiler will translate such a `main()` method into (an equivalent of) the following JavaScript code:

```
ScalaJS.c.Main$.prototype.main__V = function() {
  var playground =
    document.getElementById("playground");
  var img = new Image();
  img.src = "./path/to/img.png";
  playground.appendChild(img);
};
```

which is what you would expect. Note that `Window.document` was translated to just `document` because `Window` is a `js.GlobalScope`.

Typing callable objects. In JavaScript, function values, which can be called, are often extended with other properties and methods, giving full-fledged objects which are, in addition, callable. For example, the `jQuery` top-level object has methods like `ajax()` and properties like `fn`, but it is also callable. It can be typed accurately with a facade object that defines, among others, an `apply()` method. In Scala, calling a value is syntactic sugar for calling its `apply` method. For facade types, the compiler translates calls to the `apply` method as calling the object directly instead. jQuery would have a definition like

```
object JQuery extends js.Object {
  def getJSON(url: js.String): js.Any
  def apply(query: js.String): JQuery
  def apply(element: HTMLElement): JQuery
}
trait JQuery extends js.Object {
  def html(): js.String
  def html(v: js.String): this.type
}
```

which could be used as

```
val p = JQuery("p #playground")
p.html(p.html() + " some more text")
```

Note also how runtime-handled overloading can be described with compile-time overloading in facade types.

Typing higher-order methods. Higher-order methods can of course be typed accurately using parameters of type `js.FunctionN`. For example, jQuery objects have a method `each()` taking a function to be executed for each element in the query set.

```
trait JQuery extends js.Object {
  def each(f: js.Function2[js.Number,
      HTMLElement, _]): this.type
}
```

Scala type		JavaScript type
`Double`	↔	`js.Number`
Other numeric types	→	`js.Number`
`String`	↔	`js.String`
`Boolean`	↔	`js.Boolean`
`Array[A]`	↔	`js.Array[A]`
`FunctionN[T1,...,R]`	↔	`js.FunctionN[T1,...,R]`

Table 1. Builtin conversions for interoperability

which can be called and given a lambda as in

```
JQuery("li").each { (i: js.Number, li: HTMLElement) =>
  JQuery(li).html(i + " -> " + JQuery(li).html())
}
```

Facade types have a lot of other features enabling natural, intuitive ways of interfacing with JavaScript code. We will not cover all of them here, as they are not relevant to the purpose of this paper. A full coverage of these features is available in the documentation for Scala.js.

4.3 Type-directed conversions

Facade types allow to manipulate JavaScript values from Scala code, but they are not sufficient for true interoperability, as Scala values cannot be converted to and from JavaScript values.

Conversions are provided on a type-directed basis, using the implicit conversions introduced in Section 3.1. For example, conversions are available between `java.lang.String` and `js.String`, which allows to pass the literal `"playground"` (of type `String`) to `getElementById()` above (which expects a `js.String`). Similarly, Scala.js can convert between `FunctionN` and `js.FunctionN`, which is why it is valid to give the lambda (of type `Function2`) to the `each()` method above (which expects a `js.Function2`). These conversions are defined as

```
implicit def stringS2JS(s: String): js.String
implicit def stringJS2S(s: js.String): String
implicit def function1S2JS[T1, R](
    f: Function1[T1, R]): js.Function1[T1, R]
...
```

They do not have any body, because they are treated as primitives by the backend of the compiler. Table 1 summarizes the builtin conversions. The conversion for strings is optimized away by the code generator, because `String` and `js.String` have the same runtime representation (namely, a primitive JavaScript string). The same applies to numeric types and booleans, but not to arrays and functions.

These type-directed conversions are very much like those described in [17]. However, in Scala.js, they are more general, because they are defined by implicit conversions, and hence are user-defined.

Composing implicit conversions. The above conversions are provided as part of Scala.js, and are handled by the compiler, but developers can define their own conversions. For example, consider a data type representing an immutable point, with two fields `x` and `y`. The facade type for the JavaScript version could be defined as

```
class JSPoint(val x: js.Number,
    val y: js.Number) extends js.Object
```

whereas the Scala version would probably be defined as

```
case class Point(x: Double, y: Double)
```

Note that although similar in syntax, these two data types have very different runtime representations, for the reasons detailed in Section 4.1. Moreover, code is emitted for `Point`, but not for `JSPoint`, which is a facade class. User-defined implicit conversions can be provided with

```
implicit def pointToJS(p: Point): JSPoint =
  new JSPoint(p.x, p.y)
implicit def pointFromJS(p: JSPoint): Point =
  Point(p.x, p.y)
```

These two implicit conversions build upon the conversions between Scala doubles and JavaScript numbers. Indeed, in `pointToJS`, we give `p.x` which is a `Double` to the constructor of `JSPoint` which expects a `js.Number`. Conversions are not only user-defined, but also composable, just like data structures are composable. They are therefore a very natural way to define conversions between data structures.

Typing monkey-patching. Monkey-patching is a technique used in JavaScript and other dynamic languages to add functionality to an existing class. A well-known example of encouraged monkey-patching is jQuery plugins: the object `$.fn` is meant to be extended from the outside by plugins that want to add methods to jQuery objects. Typing statically this pattern can be non-trivial, e.g., TypeScript allows to reopen already defined interfaces to add fields and methods.

Scala has a safer abstraction (bound to lexical scope) to add features to any given type, which is also based on implicit conversions. It is sometimes known as the Pimp my Library pattern. Scala.js can effectively type monkey-patches with separate types and implicit conversions. Given `JQuery` a facade trait for jQuery objects, we can define

```
trait JQueryGreenify extends JQuery {
  def greenify(): this.type
}

implicit def jQuery2greenify(query: JQuery) =
  query.asInstanceOf[JQueryGreenify]
```

When the implicit conversion `jQuery2greenify` is in scope, calling `x.greenify()` on a `JQuery` value `x` is legal, and summons the implicit conversion. At runtime, the `jQuery2greenify` is a no-op, because calls to `asInstanceOf[T]` are erased when `T` is a facade type. This effectively calls the `greenify()` method on the original receiver `x`.

4.4 Dynamic type

Facade types and type-directed conversions allow rich, statically typed interoperability between Scala and JavaScript. However, sometimes, we do not want to, or cannot, give accurate facade types to the JavaScript API we want to use. Scala.js features a type `js.Dynamic`, which is a special facade type that can accommodate any JavaScript value. It allows to select or update any field, call any method, and apply any JavaScript operator (including the call operator) to a value of type `js.Dynamic`. It is possible to acquire a dynamically typed reference to the global scope through `js.Dynamic.global`.

Recall from the code snippet manipulating the DOM from Section 4.2. We can rewrite it using dynamic typing as follows.

```
object Main {
  def main(): Unit = {
    val g = js.Dynamic.global
    val playground =
      g.document.getElementById("playground")
    val img = js.Dynamic.newInstance(g.Image)()
    img.src = "./path/to/img.png"
    playground.appendChild(img)
  }
}
```

Most operations work seamlessly and very naturally with dynamically typed values, even though we are in a statically typed language. Instantiating an object of a dynamic type is unfortunately awkward due to technical reasons, requiring the use of a method `newInstance` instead of the familiar keyword `new`. As another example, we show how to use jQuery in a dynamically typed way to create a button that, when clicked, displays a message.

```
val jQ = js.Dynamic.global.jQuery
val button = jQ("<button>")
button.appendTo(jQ("#playground"))
button.click { () =>
  js.Dynamic.global.alert("You clicked me!")
}
```

which involves an anonymous function.

We build `js.Dynamic` on top of the `scala.Dynamic` feature of Scala, introduced in Section 3.2. We can define `js.Dynamic` as a special facade type that mixes in `scala.Dynamic`, like this:

```
trait js.Dynamic extends js.Any with scala.Dynamic {
  def applyDynamic(name: String)(args: js.Any*): Dynamic
  def selectDynamic(name: String): Dynamic
  def updateDynamic(name: String)(value: js.Any): Unit

  def apply(args: js.Any*): Dynamic

  def +(that: js.Number): js.Number
  def -(that: js.Number): js.Number
  ... // all JavaScript operator
}
```

Note that in the dynamic rewriting methods (which are considered as primitives by the backend of the compiler), input values are expected to be of type `js.Any`, and result types are typed as `js.Dynamic`. This means that, from a usage point of view, methods whose receiver is a `js.Dynamic` always expect values of type `js.Any` as arguments, and always return a `js.Dynamic`. Similarly, accessing a field is typed as `js.Dynamic`, and assigning a field expects an rhs of type `js.Any`. Hence, `jQ` and `button` are both inferred to be of type `js.Dynamic`. The fact that input values are expected to be of type `js.Any` drives the Scala type-checker into applying implicit conversions to parameters as needed. For example, when calling the `click()` method, we pass in a lambda, which has type `Function0[Unit]`. It is implicitly converted, thanks to type-directed conversions and facade types, into a `js.Function0[Unit]` which can really be given to the JavaScript API. This is necessary because `Function0` and `js.Function0` do not have the same runtime representation.

In a bit more details, when typing `g.document` in the code snippet above, with `g` of type `js.Dynamic`, the type-checker, being unable to find an explicitly defined member named `document` in `js.Dynamic`, attempts to rewrite the access as:

```
g.selectDynamic("document")
```

which succeeds, and typechecks as `js.Dynamic` according to standard typing rules. Hence, `g.document` is also typed as `js.Dynamic`. Proceeding further, we can chain the call to `getElementById` which is similarly rewritten as

```
document.applyDynamic("getElementById")("playground")
```

which is not quite well-typed yet, because `applyDynamic` expects arguments of type `js.Any`, and `"playground"` has type `String`, not `js.String`. This in turn triggers the implicit conversion from Scala string to JavaScript string, just like in the statically typed interoperability. The same applies when we call the `click` method in the jQuery example: the Scala function is converted through an implicit conversion to conform to the expected type `js.Any`. Finally, the assignment to `img.src` works with the last rewriting:

```
img.updateDynamic("src")("./path/to/img.png")
```

which also triggers an implicit conversion. Those three dynamic definitions cover all accesses to fields, and call to all methods. They are complemented with statically typed definitions for all JavaScript operators, including the call operator, represented by the `apply` method, as is the case in the statically typed interoperability.

Category	# tests	# passing	% of tests passing
"pos"	1107	1104	99.7 %
"neg"	725	720	99.3 %
"run"	1385	686	49.5 %

Table 2. Coverage of the Scala test suite

The fact that type-directed conversions and facade types play a crucial role even in the dynamically typed approach to interoperability is a unique aspect of our framework. Because type-directed conversions can be user defined, these automatic representation conversions also apply to user-defined types, such as `Point` and `JSPoint`.

5. Evaluation

We have evaluated Scala.js along three axes, which show that Scala.js is a viable, real-world alternative to JavaScript and other languages for front-end Web development.

5.1 Correctness of the compiler

We evaluate the correctness of the Scala.js compiler by having it compile the full test suite of Scala, and run the resulting code with the JavaScript interpreter Rhino. The Scala test suite consists of thousands of independent tests, in the form of one or more source files each. There are three categories of tests:

- "pos" tests: check that the test compiles,
- "neg" tests: check that the test does not compile, and that the error message is the expected one, and
- "run" tests: check that the test compiles and runs, and that the expected lines are output to the standard output.

For example, the source file for one of the "run" tests contains

```
class V(val x: Int) extends AnyVal
object Test {
  def main(args: Array[String]) = {
    val v = new V(2)
    val s: Any = 2
    println(2.getClass)
    println(v.getClass)
  }
}
```

and its expected standard output is

```
int
class V
```

Compiling the Scala file with Scala.js, then running the resulting JavaScript code, must output the lines `int` and `class V` on the standard output. This particular example tests whether `getClass()` works correctly for value classes.[2]

Table 2 shows the number of tests of the Scala test suite that pass in the current implementation of Scala.js. Very few of the non-passing tests are actual bugs. Most tests fail because they test features of Scala that are not supported by Scala.js on purpose, e.g., runtime reflection, runtime compilation, usages of the Java collection library, etc. However, quantifying exactly how many tests are legitimate is a time-consuming task that is still in progress. Our current sampling exhibits less than 2 % of legitimate tests among the failing tests, all of them related to the same bug.

[2] `getClass()` and `java.lang.Class.toString()` are part of the runtime reflection features supported by Scala.js.

Benchmark	Engine	dev (ms)	opt (ms)	js (ms)	opt/js
Tracer	V8	12,048	5,205	1,517	3.43
	Node	15,454	7,925	3,259	2.43
Richards	V8	962	431	110	4.83
	Node	1,579	706	189	3.74
DeltaBlue	V8	5,877	2,113	220	9.60
	Node	6,564	2,897	290	9.99
DeltaBlue2	V8	1,982	894	220	4.06
	Node	2,711	1,275	290	4.40

Table 3. Running times of benchmarks (less is better)

5.2 Performance

Since we argue that Scala.js can be a viable alternative to JavaScript and other languages targeting JavaScript, we have to demonstrate that the generated code is competitive with hand-written JavaScript, or at least not too much slower. The JavaScript version of the benchmarks of the Dart language [12] have been ported to Scala.js by Jonas Fonseca [10]. The methodology was to rewrite the benchmarks in idiomatic Scala code, while avoiding to deviate too much from the original code.

We executed the hand-written JavaScript code, as well as both the non-optimized and the optimized versions of Scala.js code. The non-optimized (aka "dev") version is the result of the fast development cycle. The optimized (aka "opt") version is the result of optimizing the dev version with Google Closure Compiler.

The benchmarks were done on an Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz, running GNU/Linux 3.2.0-52-generic 64-bits (Ubuntu distribution). Each test was run with the standalone interpreters of both Node.js v0.10.1 and V8 3.22.15 (candidate).

Table 3 shows the results. The last column shows the normalized running time of the opt version against the hand-written JavaScript version (js). We can see that, depending on the benchmark, Scala.js code is 2.5 to 10 times slower than hand-written JavaScript.

These results may seem unappealing, but a few things are worth taking into account. First, we have not so much as begun to investigate possible optimizations to apply to the result of Scala.js compilation. This means that there is potentially a lot of room for improvement. In particular, no inlining of any kind is performed, which is impairing a lot, because idiomatic coding in Scala tends to produce a lot of small methods with lots of indirections. Worse, all `val`s and `var`s are compiled with getters and setters (which is needed to support overriding them), but most of these accessors could be inlined with global code analysis.

The DeltaBlue benchmark performs particularly badly because it uses heavily for loops on ranges (which is known to perform badly even in standard Scala) and `ArrayBuffer`s, which are implemented in a highly suboptimal way since they were designed for Java arrays which cannot grow or shrink natively. DeltaBlue2 is another implementation of the same benchmark, which avoids those two performance killers, and thus performs more than twice better, yielding results comparable to the other benchmarks. This tells us that we could gain significant performance improvements by reimplementing `ArrayBuffer` specifically for Scala.js.

5.3 Usefulness

Our main claim throughout has been that the interoperability system in Scala.js is natural, intuitive, and more importantly, expressive enough to interact with existing JavaScript APIs. In other words, we claim that our interoperability framework is *useful*.

To support this claim, we have ported to Scala.js a few applications, previously written either in JavaScript, Java, or Scala. In the process, we have used facade types written for the DOM and for jQuery, as well as the `js.Dynamic` type, thereby heavily exercising the interoperability framework of Scala.js.

Typing the DOM. Facade types for the DOM API have been made available by Haoyi Li, based on a rough automated source-to-source translation from TypeScript type definition files, for a total of about 5,000 lines of code. These facade types have been used in six applications written in Scala.js, with a combined size of about 1,000 lines of code. In this code, exactly two casts are required. The first one casts the result of fetching an HTML element on the page by its id to its actual element class:

```
val canvas = document.getElementById(
    canvasName).asInstanceOf[HTMLCanvasElement]
```

Depending on the DOM tree itself, this cast is unavoidable in most type systems, unless support for the DOM tree is included in the type system itself, as recently proposed by Lerner et al. for jQuery [16]. The other cast required specializing the type of graphics context retrieved from the canvas:

```
val ctx = canvas.getContext(
   "2d").asInstanceOf[CanvasRenderingContext2D]
```

The method `getContext()` returns a different type of context depending on the value of the string argument. Encoding this in the type system requires either dependent types, or a very weak special-case based on overloading the method type on constant strings, which we could write as follows:

```
def getContext("2d"): CanvasRenderingContext2D
def getContext("webgl"): WebGLRenderingContext
```

This weak form has been implemented in TypeScript, and is considered for Scala.js in future work. In addition to the few number of casts, no explicit invocation of cross-language conversions is required. These measurements show that our framework can type accurately large amounts of the DOM API, allowing to interact naturally with all of the DOM with very few casts.

Typing jQuery. jQuery makes heavy use of higher-order methods and runtime overloading, compared to the DOM, which exercises more inheritance relationships. Facade types for jQuery have been used similarly in two applications, of around 250 and 500 LoCs, respectively.

As was the case for DOM manipulations, only two casts were required: the same exact two casts, as it turns out.

Using js.Dynamic. All the previous programs can be rewritten to use only `js.Dynamic`-based interoperability. In this case, entry points (fetching the global scope, and instantiating new objects) have a less nice syntax, as we saw when presenting `js.Dynamic`.

Compared to the statically typed versions, using `js.Dynamic` for interoperability only adds casts where values must be converted back to Scala values. A representative example is the following, where coordinates of a click event are retrieved:

```
boardCanvas.click { (event: js.Dynamic) =>
  val offsetX = (event.pageX -
      boardCanvas.offset().left)
  val x = (offsetX.asInstanceOf[js.Number].toInt /
      SquareSizePx)
  ...
  clickSquare(board(x)(y))
}
```

In our corpus, these JavaScript to Scala conversions happened to be scarce, with about 20 casts needed. They all correspond to fetching input values coming from DOM elements or DOM events. Scala to JavaScript conversions never required any cast nor explicit conversions.

6. Related work

The three features of our interoperability framework, facade types, type-directed conversions and the Dynamic type, although never brought together in a unified, coherent framework, are variations of features studied informally and formally by various authors. In this section, we review similar work to each of them, and highlight the differences with our variant and how they contribute to unifying the framework.

6.1 Facade types

Variants of what we have called facade types, similar in spirit in that they give static types to values for the purpose of interoperability, have existed in a few recent, preliminary works.

In SMLtoJs [9], Elsman proposes to use phantom types to represent JavaScript values. However, they do not provide first-class access to the methods of such values. Instead, bridge functions must be manually written with embedded strings of JavaScript code, with manual bindings for the `this` objects and arguments to the method. Facade types of Scala.js are truly first-class and allow direct method invocation, eliminating the need for manual bridges, and thus providing a more natural access to JavaScript APIs.

The concept closest to facade types is found in the TypeScript programming language [18], although they are not used for interoperability. TypeScript does not need interoperability at all, since it belongs to the “It is just JavaScript” category of languages. However, TypeScript aims at providing static types to JavaScript, and hence needs type information for JavaScript APIs, which can be written in type definition files, as ambient definitions. The ambient definitions of TypeScript have inspired the facade types of Scala.js. However, a few differences exist, most of them being non-fundamental, simply a better integration with Scala’s type system in general. A key difference is our distinction between `js.Any` and `js.Dynamic`. In TypeScript, the top-level type `any` is also the equivalent of our dynamic type, as it supports all operations. This is clearly unsound, as any value of type `T`, e.g., `DOMDocument`, can be first assigned to a value of type `any`, then used as a dynamic type. TypeScript embraces unsoundness, and this decision was motivated by one of their mottos which is that any JavaScript program also typechecks as a TypeScript program. Free of this constraint, we chose a more type-safe path in which `js.Dynamic` is a subtype of `js.Any`. Explicit casts to `js.Dynamic` can be done if necessary, but they make the intent explicit.

Finally, we relate facade types to like types as introduced and formalized by Wrigstad et al. [28]. Like types have been proposed to progressively add types to an untyped language (which is also the goal of other approaches, most prominently gradual typing). A like type `like C` can be instantiated for any class (or trait if we generalize the concept for Scala). When accessing a member of some value `x` of type `like C`, the type system requires that the member exist in class `C`. This avoids spelling mistakes and other type errors in a code annotated with like types. However, it is always possible to assign a value of any type to a value `x` of type `like C`. Since this makes access to members of like types unsound, they are protected by dynamic checks at runtime (whereas accesses to members of full types can be optimized statically). Although facade types have a very different goal, namely interoperability, they share a surprisingly large amount of properties with like types. Any value can be assigned to a facade types, although this requires an explicit cast (but the cast is not checked at runtime), so accesses to members of facade types are not completely sound. However, they are inherently protected by the underlying JavaScript interpreter, just as like types are. Conversely, full types (Scala.js-emitted classes and traits) are sound, since casts to such values are checked at runtime. Therefore, a better optimizer could optimize accesses to Scala.js members, for example by static inlining. Since like types have been formalized, they could provide inspiration for a formalization of our interoperability framework.

6.2 Type-directed conversions

Type-directed conversions for interoperability have been studied extensively by several authors [1, 23], and formalized recently by Matthews and Findler [17]. In these works, however, the expected type of the conversions from dynamically typed to statically typed must be specified explicitly, and dynamic type checks are inserted to verify the assumption. The corresponding frameworks, lacking facade types, only support a dynamic type for values of the dynamically typed language. They are in effect almost equivalent to having only type-directed conversions and `js.Dynamic` in our framework. The combination with facade types allows most conversions from JavaScript to Scala to be entirely implicit, which simplifies interoperating spots. Leveraging implicits as available in Scala and formalized in [22] also allows for recursive conversion functions.

6.3 Dynamic type

The presence of a dynamic type in a statically typed language is a common property of several recent languages, such as Thorn [2] and Sage [14]. However, they are not used for interoperability, and hence do not need any combination with cross-language conversions. The `js.Dynamic` type of Scala.js is inspired by these instances of dynamic types, but builds upon the statically typed information provided by facade types and type-directed conversions to perform transparent, automatic cross-language conversions as needed. Because methods of `js.Dynamic` take inputs of expected type `js.Any`, and not `Any` as typical dynamic types, they trigger the user-defined cross-language conversions available in scope. Hence, our dynamic type is fully unified with the two other aspects of our framework.

Our `js.Dynamic` type, interoperability concerns put aside (including implicit conversions), can also be related to the dynamic type of gradual typing (denoted ?), and in particular as studied by Siek and Taha in the calculus $\mathbf{Ob}^{?}_{<:}$ [24]. As is the case in $\mathbf{Ob}^{?}_{<:}$, the `js.Dynamic` type is different from `js.Any`. However, we have decided not to mimic, by default, the implicit coercions from ? to any type and vice versa. Such coercions can instead be written explicitly in Scala.js using `asInstanceOf`. Note that, because implicit conversions are user-defined, developers can define custom implicit conversions from/to `js.Dynamic` for some or all types, as needed. If the implicit coercions were introduced by default, they could not be disabled, hence our choice. The other major difference between our framework and $\mathbf{Ob}^{?}_{<:}$ is that we do not perform runtime checks when down-casting or invoking methods of `js.Dynamic`. This is in line with the general semantics of `asInstanceOf` for JavaScript values, which is not to perform runtime checking of the cast.

7. Conclusion

Interoperability between statically typed and dynamically typed languages is an important problem. In this paper, we have presented a framework for type-directed interoperability with dynamically typed languages, which provides very powerful and natural interoperability. The framework is built on facade types, type-directed implicit cross-language conversions, and a dynamic type which is also type-directed.

Facade types provide type information to APIs and data types written in the dynamically typed language. Type-directed, user-defined, implicit conversions provide automatic conversion of runtime data representation. They can be defined by the user to support custom data types. Finally, the dynamic type allows to manipulate values of the dynamically typed language in a whitebox way without requring dedicated facade types. It builds on top of facade types and type-directed conversions to provide the same runtime data representation conversions.

We have implemented our framework in the Scala.js language and its compiler, which is a dialect of Scala compiling to JavaScript. We showed how the framework can be implemented on top of the existing type system of Scala, without modifying it, by using implicit conversions and custom dynamic types. Any language supporting these two features could benefit directly from our framework, without changing its typing rules. We have evaluated the compiler and the approach for correctness, performance, and most importantly usefulness of the interoperability framework, thereby demonstrating that it can accurately type the DOM and jQuery APIs.

Acknowledgments

We thank Jonas Fonseca for his ports of the Dart benchmarks to Scala.js and Haoyi Li for his comments about the practical usability of Scala.js.

References

[1] N. Benton. Embedded interpreters. *J. Funct. Program.*, 15(4):503–542, July 2005.

[2] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the jvm. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 117–136, New York, NY, USA, 2009.

[3] E. Burmako and M. Odersky. Scala macros, a technical report. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.

[4] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 278–292, New York, NY, USA, 1991.

[5] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 587–606, New York, NY, USA, 2012.

[6] ClojureScript. ClojureScript, 2013. URL `http://clojure.org/clojurescript`. [Online; accessed 29-October-2013].

[7] CoffeeScript. CoffeeScript, 2013. URL `http://coffeescript.org/`. [Online; accessed 25-October-2013].

[8] ECMA International. ECMAScript 5.1 language specification, June 2011. URL `http://www.ecma-international.org/ecma-262/5.1/`. [Online; accessed 25-October-2013].

[9] 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, New York, NY, USA, 2011.

[10] J. Fonseca. Scala.js benchmarks, 2013. URL `https://github.com/jonas/scalajs-benchmarks`. [Online; accessed 25-October-2013].

[11] Google. Google Closure compiler, 2013. URL `https://developers.google.com/closure/compiler/`. [Online; accessed 29-October-2013].

[12] Google. Dart, 2013. URL `https://www.dartlang.org/`. [Online; accessed 30-October-2013].

[13] Google. Google Web Toolkit, 2013. URL `http://www.gwtproject.org/`. [Online; accessed 29-October-2013].

[14] J. Gronsky, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop (Scheme)*, pages 93–104, Sept. 2006.

[15] D. Herman and C. Flanagan. Status report: specifying JavaScript with ML. In *Proceedings of the 2007 workshop on Workshop on ML*, ML '07, pages 47–52, New York, NY, USA, 2007.

[16] B. S. Lerner, L. Elberty, J. Li, and S. Krishnamurthi. Combining form and function: static types for jQuery programs. In *Proceedings of the 27th European conference on Object-Oriented Programming*, ECOOP'13, pages 79–103, Berlin, Heidelberg, 2013.

[17] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3):12:1–12:44, Apr. 2009.

[18] Microsoft. TypeScript, 2013. URL `http://www.typescriptlang.org/`. [Online; accessed 25-October-2013].

[19] A. B. Nielsen. Scala compiler phase and plug-in initialization, 2008. URL `http://www.scala-lang.org/old/sid/2.html`. [Online; accessed 29-October-2013].

[20] M. Odersky. The Scala language specification, 2009. URL `http://www.scala-lang.org/documentation/`. [Online; accessed 7-November-2013].

[21] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010.

[22] B. C. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The implicit calculus: a new foundation for generic programming. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 35–44, New York, NY, USA, 2012.

[23] N. Ramsey. Embedding an interpreted language using higher-order functions and types. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, IVME '03, pages 6–14, New York, NY, USA, 2003.

[24] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73588-5.

[25] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 793–810, New York, NY, USA, 2012.

[26] P. Thiemann. Towards a type system for analyzing javascript programs. In M. Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 408–422. Springer Berlin Heidelberg, 2005.

[27] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 395–406, New York, NY, USA, 2008.

[28] 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, New York, NY, USA, 2010.