École Polytechnique Fédérale de Lausanne

Master Semester Project

Production-level tooling for the Kotlin to Scala.js compiler

Author:
Florian Alonso

Professor:
Martin Odersky

Supervisor:
Sébastien Doeraene

January 12, 2018

LAMP

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Acknowledgements

I would like to thank my advisor, M. Sébastien Doeraene, for all the support he provided me with and for the advice he gave me during this semester of work. His help has been very valuable to me and has allowed me to learn many things along the way.

A huge thank you to my friend Dennis as well, who helped me find inconsistencies in this report by reading through it in time of exams.

Thank you finally to the EPFL and LAMP for giving me the opportunity to work on this project.

Contents

Chapter 1

Introduction

1.1 Kotlin and Scala.js

JavaScript (JS for short) is a programming language that provides developers with a way to interact with users by building dynamic web applications and websites. Through the use of JavaScript runtimes, such as Node.js [4], it is no longer required that applications be executed only on the client side. It can therefore also be used to build a large range of server applications. Many frameworks and libraries exist to provide useful functionalities to the language and ease the development of said applications.

JavaScript is a weakly typed language and provides a large freedom on how to organize one's code. However, depending on the situation, a strongly typed language might be more suitable for development. This is where Kotlin JS and Scala.js come into play.

Kotlin JS Kotlin [23] is a statically typed programming language providing many features through its large standard library. It is able to run on the JVM and is officially supported for Android since May 2017 [3]. In order to expand their reach, the Kotlin team has built a JavaScript compiler to translate the Kotlin standard library and the language capabilities into a JS application. It however lacks some optimizations and produces a rather large amount of JS code.

Scala.js Scala.js [5] is another typed programming language. Being implemented as a Scala compiler plugin, it benefits from most of its features with a few differences due to the JavaScript semantics (more details about this can be found in the "Hands on Scala.js" tutorial [2]).

Both languages provide specific language features in order to allow developers to interact with pre-existing JavaScript objects and thereby provide JavaScript interoperability. The main difference between Kotlin JS and Scala.js is the number of optimizations performed on the code before outputting JavaScript. The sizes of the final JS files are also very different.

Scala.js (mainly thanks to its linker) enhances the performance by performing whole program optimizations and greatly reduces the size of the generated code. All details about the Scala.js compilation pipeline (and how it optimizes JavaScript code) can be found under section “The compilation pipeline” of the “Hands on Scala.js” [2] tutorial. Kotlin, since its version 1.1.4, provides Dead Code Elimination through an additional Gradle plugin which can be found in the Kotlin documentation [20].

As a summary, combining both the Kotlin language flexibility and the Scala.js optimizing capabilities can result in a great improvement for JavaScript applications development both in terms of executable size and performance.

1.2 Motivations

I have always been strongly interested in web related technologies and the opportunity to work at the core of one of them is really an enriching experience. The Kotlin compiler is a very large code base and having to search trough its numerous source files in order to understand the logic behind the language is challenging and at the same time captivating.

Besides the performance aspect behind the compilation pipeline (and of course the language syntax), one of the most important things for a language to be adopted is the tooling available and how easy it is to setup a new project and compile one’s code.

The main goal of this project was thus to provide developers with basic tooling and support for the Kotlin language. This will be achieved by continuing prior work to compile the Kotlin standard library to the Scala.js IR and by writing a simple Gradle [10] plugin.

1.3 Project structure and architecture

The work on a Kotlin to Scala.js compiler was started in February 2017 by Lionel Fleury and Guillaume Tournigand. Their work resulted in a structured code base which allowed simple Kotlin programs to compile (for more details on their work, please refer to their semester project report [30]). A brief review of the working features of their project is available under section 2.1.

Main source code

The main code of the project is stored under the `ch.epfl.k2sjsir` (root) package, organized in three subpackages: `lower`, `translate` and `utils`.

More details about this design and the underlying Kotlin AST (PSI, Descriptors and Contexts) can be found under section "Kotlin" of Guillaume's and Lionel's report [30].

Package root A few files are located at the root of the `ch.epfl.k2sjsir` package and they are the entry point into the Kotlin compiler pipeline. For reference, the `translate` package is first called via the `GenClass` class from the `PackageDeclarationTranslator` file.

Package `lower` This package contains all the logic related to class lowering which is used to retrieve inner classes and objects (besides anonymous objects, see section 2.10) from the Kotlin AST.

Package `translate` This is where the main translation logic is kept. It is organized in a generator pattern and thus every class in this package must provide a way to retrieve an SJSIR object (generally through a `tree` method) from a `KtElement`. For example if you are provided with a `KtExpression` object, you can translate it with:

```
val e: KtExpression = /* Retrieve a KtExpression */
GenExpr(e).tree // This will return a Scala.js IR node
```

Package `utils` This last package contains a variety of utility methods such as implicit classes which provide an interface to encode names, retrieve types or generate recurring SJSIR objects, etc.

Tests

In order to verify that no code change breaks the translation of other elements, a test suite exists under the `test` directory. This directory is organized in two main folders which are `resources` and `scala`.

Folder `resources` There are mainly two directories in this folder; the first one, `lib`, contains the Scala.js library jar file. These are used when launching the Scala.js linker to compile the SJSIR files output by the compiler. The second folder is the `src` folder which contains all Kotlin sources to test the good behavior of the Kotlin to Scala.js compiler.

Folder `scala` It contains the various ScalaTest [7] files which define the unit tests. Each test file was written to test a specific language feature, please refer to section 2 for details about which Kotlin feature these test files are related to. The corresponding source code can be found in the GitHub repository [29].

General design of the project

Entry point The `K2SJSIRCompiler` class is used in order to inject the Scala.js IR translation logic into the Kotlin compiler. It is where the parsing of the command line arguments happens, through the use of the `K2SJSIRCompilerArguments` class. The resolution of source file paths is then handed to the `K2SJSIRTranslator` which in turn will delegate the work to the `Translation` class.

Finally, this class will hand the task of generating SJSIR nodes to the `PackageDeclarationTranslator` class. It then makes use of the content of the `translate` package in order to generate the final SJSIR files.

Since this code is (mostly) a Scala translation of the original `K2JSCompiler` class, all phases (parsing, type checking, etc.) of the original compiler are still there.

Translation As explained above, the translation happens mainly inside the `PackageDeclarationTranslator` class. It will extract the various declarations using lowering and generate the corresponding SJSIR files. The declarations can be either classes and objects or top-level functions and properties.

The reasoning behind this design is explained in more details in the “Hello World” section of Guillaume and Lionel’s report [30].

Chapter 2

Supported features

2.1 Prior work

This section aims to list the features supported by the prior version of the compiler [31] (as of July 2017). This version was built to work with Scala.js 0.6.15 and Kotlin JS 1.1.1. Since then, it was upgraded to Scala.js 1.0.0-M2 and Kotlin 1.1.61 [18].

Below is a list of all features that were already working (or after minor fixing) on the version of the compiler this project started with. For specific syntax examples please refer to the Kotlin documentations available online [24].

Top-level and extension functions These are translated the way Kotlin does it. This means these definitions are stored in a class named after their containing source file. Top-level properties on the other hand are not supported yet.

For example, if a function `fun dummyFun() = println("Hello World")` is in a source file called `HelloWorld.kt`, it will be stored in a class called `HelloWorldKt` as a `static` definition which is in turn stored in an SJSIR file.

Classes and inheritance "Classic" classes (as opposed to `data`, `enum`, `annotation`, etc.) are well supported. They can be declared (and instantiated) with :

- A primary constructor and zero or more secondary constructors.
- Inheritance from a superclass (if the class is marked as `open`).

- One or more `init` blocks.
- Methods and properties (overridden or not); for properties you couldn't define custom getters and setters. However the default ones were generated properly, see section 2.6.
- Access to the current class properties and methods. You couldn't however use `this` nor `super` (see sections 2.3 and 2.4).

In the case of `inner` classes, the behavior is the same as for nested classes. That is, calls to methods and properties of the enclosing class are not supported yet.

Classes are stored within an SJSIR file with the same name as the class itself. For example `class Customer` would be stored in file `Customer.sjsir`.

Objects Objects and companion objects are supported at the same level as classes. The major difference being that they are generated as module classes (in the Scala.js meaning). Their SJSIR file will be named either after the object itself, prefixed with the enclosing class name and suffixed with a $. For example `object MyObject` would be written to `MyObject$.sjsir` and the following object declaration inside `MyClass` would be contained in a file named `MyClass$MyCompanion$.sjsir` as shown in figure 2.1.

```kotlin
class MyClass {
  companion object MyCompanion {
  }
}
```

Figure 2.1: A companion object defined in Kotlin

Primitive types All primitive types are supported and mapped to the corresponding Scala.js IR type. `Char` was however not fully supported. String templates worked fine, for example :

```kotlin
val name = "World"; val s: String = "Hello $name"
```

Operators Unary and binary operators are supported. Primitive type conversion of the form `toX()` were not fully available. Only structural equality was missing. Operator overloading is not supported. The reference operator `::` works for function references but hasn't been thoroughly tested.

Nullable types One could declare and use nullable types inside the code. There was however no support for safe operators (see section 2.12).

Arrays The Kotlin `Array` type worked, as well as the special functions `arrayOf(elements)`, `arrayOfNulls(size)`. Other types such as `IntArray`, `DoubleArray`, etc. are not supported. In the current version of the compiler, these types are translated to their `Array<Type>` equivalent.

Control flow Expressions such as `if`, `when`, `for` and `while` are supported. The `do {/* code */} while (/*condition */)` expression was however not implemented.
The expressions `for(v in values) {/*code using v */}` for an array of values and `for(i in 0..X) {/*code using i*/}` for `X` an integer expression worked as well but one could not use neither `until` and `downTo` nor `step`. These are now supported.

Variables and values Variables and values can be declared inside classes or functions. If a `var` is declared, then it can be modified with an assignment operator. For example :

```
fun main(args: Array<String>) {
  var counter = 0
  counter = 1
  counter++
  counter += 1

  val finalCounter = counter + 1
  println(finalCounter) // prints "4" as expected
}
```

Interaction with JavaScript Accessing to existing JavaScript objects and instances is supported in a limited way through the use of the `external` keyword. This allows for `external class` declarations with member functions. Properties are however not supported and dynamic types are not properly handled yet.

2.2 Type checks and casts

The keywords `is`, `!is` and `as` are supported. The Kotlin smart cast feature is however not yet implemented and requires casting variables manually to use them as an instance of the checked type :

```kotlin
class MyClass { fun doNothing() = println("Doing nothing") }

fun checkFromAny(a: Any) {
  if (a is MyClass) {
    val tmp = a as MyClass // no smart cast
    tmp.doNothing()
  } else if (a !is String) {
    println("$a is not a string !")
  }
}

fun main(args: Array<String>) {
  checkFromAny(MyClass()) // prints "Doing nothing"
  checkFromAny(42) // prints "42 is not a string"
}
```

The `is` operator is translated to the Scala.js `IsInstanceOf` IR node. If the `!` is used, it will then be translated as a unary operation negating the result of the check.

The `as` operator on the other hand is translated either to an `Unbox` or an `AsInstanceOf` IR node. The latter is used if the cast expression is neither a primitive type nor of type `Unit` or a lambda expression (see section 2.7 for details about lambdas).

2.2.1 Related files

The main source files involved in the translation of these two operators are the following.

- `translate/GenIs.scala`
- `translate/GenExpr.scala` (look for match cases `KtIsExpression` and `KtBinaryExpressionWithRHS`)
- `utils/Utils.scala` (look for the `cast` function)

Test files The test file verifying that these features are behaving as expected is as follows :

- `TestIsOperator.kt`

2.3 Qualified `this`

The keyword `this` is now supported. The way it is handled depends on the context where it is used. In classes it is simply translated to the `This` IR node, meanwhile in extension functions or in interfaces default implementations it is translated to `VarRef(Ident("$this"))` which corresponds to the receiver parameter added to the function arguments (see section 2.8 about interfaces as well).

For example, the code given in figure 2.2 translates to the printed IR representation given in figure 2.3.

```
// Extension function definition
fun String.printWith(s: String) {
  println(this + " " + s)
}
```

Figure 2.2: Definition of an extension function

```
// Result in the IR representation
static def printWith__T__T__V($this: T, s: T) {
  mod:s_Predef$.println__O__V(
    ((("" +[string] $this)
    +[string] " ") +[string] s)
  );
  (void 0)
}
```

Figure 2.3: Translation of an extension function

2.3.1 Related files

The main source files involved in the translation of this keyword are the following:

- `translate/GenExpr.scala` (look for match case `KtThisExpression`)
- `translate/GenFun.scala` (look for the `tree` function)
- `translate/GenProperty.scala` (look for the `getter` function)
- `utils/Utils.scala` (look for the `genThisFromContext` functions)

Test files The test file verifying that these features are behaving as expected is as follows.

- `TestThisKeyword.kt`

2.4 Super calls

The `super` keyword has no direct correspondence in the Scala.js IR. It has different translations depending on whether it refers to a parent class or to an interface.

Super calls to a parent class In such a case the Scala.js linker expects an `ApplyStatically` node to be able to link to the correct method.

In order to be able to access the fields of a parent class, the naming of the inner backing field has been changed. This means that if a class `MyParent` has field `val a`, then if the class `MyChild` overrides this field, the inner representation of those two fields will be different. This allows `MyChild` to call `super.a` and get the original field value. This example is illustrated in figure 2.4 with its corresponding SJSIR result in figure 2.5.

The naming convention used when generating fields is a simple suffixing of the original name with the number of parent classes of the current class. In the previous example `MyParent` would therefore have a field `a` and `MyChild` would have `a$1` (note that the Object class is not taken into account).

```
interface AnInterface {
  val a: String
    get() = "Hello from parent interface"
}

open class MyParent {
  open val a: String = "Hello from parent class"
}

class MyChild: MyParent(), AnInterface {
  override val a: String = "Hello from child"

  fun example() {
    // prints "Hello from parent interface"
    println(super<AnInterface>.a)
    // prints "Hello from parent class"
    println(super<MyParent>.a)
    // prints "Hello from child"
    println(this.a)
  }
}
```

Figure 2.4: A super call to a parent class and an interface's default implementation

Super calls to an implemented interface If an `interface` has a default implementation, the programmer can call this default method from the implementing class. Because of the way these default implementations are handled, an `ApplyStatic` node is necessary (see section 2.8 about interfaces). This translation is illustrated in figures 2.4 and 2.5 as well.

The behavior of the compiler has not been tested in the context of `inner` classes, as those aren't supported yet.

2.4.1 Related files

The main source files involved in the translation of super calls are the following:

- `translate/GenExpr.scala` (look for match case `KtSuperExpression`)

```
class LMyChild extends LMyParent implements LAnInterface {
  val a$1: T
  def a__T(): T = {
    this.a$1
  }
  def example__V() {
    mod:s_Predef$.println__O__V(
      LAnInterface$DefaultImpls::a__LAnInterface__T(this)
    );
    mod:s_Predef$.println__O__V(this.LMyParent::a__T());
    mod:s_Predef$.println__O__V(this.a__T());
    (void 0)
  }
  def init___() {
    this.LMyParent::init___();
    this.a$1 = ("" +[string] "Hello from child")
  }
}
```

Figure 2.5: Super calls inside `MyChild` translated to the SJSIR

- `translate/GenCall.scala` (look for the `isSuperCall` variable)

Test files The test files verifying that these features are behaving as expected are the following.

- `classes/TestClasses.kt`
- `classes/MyClassWithSuper.kt`

2.5 Unary and binary operations

These operations were, for the most part, already supported when this project started. Support for the unary operations `toChar()` and `char.toX()` has been introduced with this new version and is translated to a `UnaryOp` SJSIR node. The structural equality, represented as `==` which was previously missing has been added as well, see figure 2.6.

```
val a = Dummy()
val b = Dummy()

println(a == b) // prints false

// Translates to
val a: LDummy = new LDummy().init___();
val b: LDummy = new LDummy().init___();
mod:s_Predef$.println__O__V({
  val x$1: LDummy = a;
  val x$2: LDummy = b;
  if ((x$1 === null)) {
    (x$2 === null)
  } else {
    x$1.equals__O__Z(x$2)
  }
});
```

Figure 2.6: Structural equality and its SJSIR translation

Since the compiler was upgraded from Scala.js 0.6.15 to 1.0.0-M2, the way binary and unary operations are handled has been changed significantly. A utility function `adaptPrimitive` was ported from the Scala.js code and is used to generate the `toX()` unary operations on primitive types.

The main code handling binary operations first makes use (if necessary) of this `adaptPrimitive` function and selects the corresponding `BinaryOp.Code` before returning a `BinaryOp` SJSIR node.

As of this writing, the compiler doesn't support operator overloading for user defined classes.

2.5.1 Related files

The main source files involved in the translation of operations are the following.

- `translate/GenUnary.scala`
- `translate/GenBinary.scala`

- `translate/GenCall.scala` (look for `isUnaryOp`)
- `utils/Utils.scala` (look for the `adaptPrimitive` function)
- `org.scalajs.core.ir.Trees` (look for `UnaryOp` and `BinaryOp`)

Test files The test files verifying that these features are behaving as expected are the following.

- `TestMethodsBaseTypes.kt`
- `TestBinaryOps.kt`

2.6 Properties accessors

Kotlin supports two types of properties which are either `var` or `val` declarations. By default, a getter (and a setter if it's a `var`) are generated by the compiler.

Backing field If at least one of the two default accessors is used or if the `field` keyword is used, the Kotlin compiler generates what is called a *backing field*. This field will be initialized to the value set by the programmer and will be accompanied by the corresponding accessors. Figures 2.7 and 2.9 illustrate this with two code examples taken from the Kotlin documentation [24].

```
var counter = 0
  set(value) {
      if (value >= 0) field = value
  }
```

Figure 2.7: An example of code with a backing field generated

When no backing field is generated, the expression contained either in the defined accessor or on the right hand side of the property definition is directly written inside the corresponding accessor as illustrated by figure 2.10.

```
var counter: int
def counter__I(): int = {
  this.counter
}
def counter__I__V(value: int) {
  if ((((double)value) >=[double] ((double)0))) {
    this.counter = value
  };
  (void 0)
}
```

Figure 2.8: The SJSIR of `var counter` with default getter and custom setter

```
val isEmpty: Boolean
  get() = this.size == 0
```

Figure 2.9: An example of code without the generation of a backing field

```
def isEmpty__Z(): boolean = {
  (this.size__I() ==[int] 0)
}
```

Figure 2.10: The SJSIR of `val isEmpty` with no backing field and custom getter

Accessors generation Accessors must be generated from different kind of `KtElement`, mainly `KtProperty` and `KtParameter` objects. The latter represents primary constructor parameters of a class for which, if marked as `val` or `var`, accessors must be generated. The `KtProperty` on the other hand represents either class properties or top-level properties. Its properly supported in the context of classes but is however not yet properly translating top-level properties.

In order to keep the same logic for both cases, the `GenProperty` class makes use of overloading in order to retrieve the common descriptors of each to finally delegate the generation of both default and custom accessors to the same methods, namely `getter` and `setter`.

For the special case of accessors inside interface default implementations, please refer to sections 2.3 and 2.8.

2.6.1 Related files

The main source files involved in the translation of accessors are the following.

- `translate/GenProperty.scala`
- `translate/GenClass.scala` (look for `GenProperty` usage)
- `utils/GenClassUtils.scala` (look for `GenProperty`)
- `utils/Utils.scala` (look for the `getMemberDefinitions` function)

Test files The test file verifying that accessors are generated as expected are the following.

- `TestAccessorsGen.kt`

2.7 Higher-order function and lambdas

As shown by figures 2.11, 2.12 and 2.13, Kotlin provides different ways of defining lambda expressions which can be passed as parameter to higher-order functions.

```
fun plus(a: Int): (Int) -> Int  = { b ->
  a + b
}

// Calling it
plus(3)(7) // returns 10
```

Figure 2.11: A lambda `(Int) -> Int` returned by a function taking one `Int` parameter

```
var times: (Int, Int) -> Int = { a,b ->
  return a * b
}

// Calling it
times(2, 3) // returns 6
```

Figure 2.12: An `(Int, Int) -> Int` lambda expression stored inside a `var`

```
fun mapFive(f: (Int) -> String) = f(5)
// Calling it
mapFive({x -> "I contain $x"}) // returns "I contain 5"
```

Figure 2.13: An anonymous lambda function

A simple implementation using the `sjsr_AnonFunction` from Scala.js was already in place in order to provide support for anonymous lambda expressions. This implementation was changed to add support for the two other examples and to be closer to JavaScript semantics. These expressions are therefore directly contained inside a Scala.js `Closure` IR node, capturing the parameters available from a declaring function (see example 2.11).

Behind the scenes, variables are captured by the closure parameters as shown in figures 2.14 and 2.7. This means they are assigned to a closure parameter and defined again inside the closure itself.

```
def plus__I__O(a: int): any = {
  (lambda<$this: LDummy = this, a: int = a>(closureargs$b: any) = {
    val b: int = closureargs$b.asInstanceOf[I];
    (a +[int] b)
  })
}
```

Figure 2.14: The SJSIR equivalent of figure 2.11

Calls to lambdas are translated by the Kotlin compiler to a call to the `invoke` function (with zero or more parameters depending on the the lambda

```
// Inside the init call of a class (accessors are not shown)
this.times = (lambda<$this: LDummy = this>(closureargs$a: any,
  closureargs$b: any) = {
  val a: int = closureargs$a.asInstanceOf[I];
  val b: int = closureargs$b.asInstanceOf[I];
  (a *[int] b)
})
```

Figure 2.15: The SJSIR equivalent of figure 2.7

declaration). This call will be translated as a `JSFunctionApply` Scala.js IR node.

It is relevant to note that these nodes do not take a type argument. This is because they map directly to JavaScript and are therefore of type `Any`. As a result, calls must be cast to the expected type in order for the linker to be able to do its work. The casting logic is contained inside the `cast` function from the `Utils.scala` file.

As of this writing, variables declared before or in the same scope as the lambda expression cannot be used inside the lambda itself because they are not captured. This problem can be solved using free variables analysis on the closure. If time permits, this should be fixed in a later version.

2.7.1 Related files

The main source files involved in the translation of lambda and closures are the following.

- `translate/GenExpr.scala` (look for `genLambda` and `genClosure` functions)
- `translate/GenCall.scala` (look for `JSFunctionApply` usage)
- `utils/Utils.scala` (look for the `cast` function)

Test files The tests verifying that lambdas and high order functions work as expected are found in the following files.

- `TestHighOrderFunction.kt`
- `TestLambda.kt`

2.8 Interfaces

As shown in figure 2.16, interfaces in Kotlin allow developers to provide a default implementation. This default method will be used if an implementing class doesn't provide its own override.

```kotlin
interface MyInterface {
  val a: String
    get() = "Hello from an interface"
}

class MyClass: MyInterface

fun main(args: Array<String>) {
  MyClass().a // returns "Hello from an interface"
}
```

Figure 2.16: An interface with default implementation of an accessor of `a`

Because we cannot store the default implementation directly inside the interface, the compiler will generate (as does Kotlin) a second file suffixed with `$DefaultImpl`. This file contains static methods, the body of which was provided as the default implementation (see figure 2.17).

```
class LMyInterface$DefaultImpls extends O {
  static def a__LMyInterface__T($this: LMyInterface): T = {
    ("" +[string] "Hello from an interface")
  }
}
```

Figure 2.17: A default implementation in the Scala.js IR

Bridging If no override is provided inside the implementing classes, the compiler will generate bridging methods. These are standard methods that act as a bridge between the implementing class and the default implementation of the interface. This means that they will statically call the corresponding default implementation as illustrated in figure 2.18.

```
class LMyClass extends O implements LMyInterface {
  def a__T(): T = {
    LMyInterface$DefaultImpls::a__LMyInterface__T(this)
  }
  def init___() {
    this.O::init___()
  }
}
```

Figure 2.18: The bridging method inside `MyClass` in the Scala.js IR

Accessors The behavior for (default implementation) accessors differs from the generation detailed in section 2.6 in that they will all receive an instance of the implementing class as first argument. The calls on `this` are therefore translated to calls on this instance parameter. Again, bridges for these accessors will be generated only if no override is provided in the implementing class.

It is important to note that backing fields are not allowed in interfaces, therefore the default implementations cannot provide custom setters which use the `field` keyword.

Finally, the interface itself is written as a class with only abstract methods. The resulting IR for `MyInterface` is shown in figure 2.19.

```
interface LMyInterface {
  def a__LMyInterface__T($this: LMyInterface): T = <abstract>
}
```

Figure 2.19: The representation of `MyInterface` in the Scala.js IR

2.8.1 Related files

The main source files involved in the translation of interfaces are the following.

- `translate/GenClass.scala` (look for `treeDefaultImpls` method)
- `translate/GenClassUtils.scala` (look for `getDefaultImplementations` and `getInterfaceBridges`)

- `utils/Utils.scala` (look for the `isInterface` keyword)

Test files The tests verifying that interfaces are implemented as expected are all found in the following files.

- `interfaces/<*>.kt`

2.9 Enum classes

Enum classes differ from "normal" classes in that they have a somewhat different representation in the Kotlin AST. The values of the enumeration are instances of classes represented by specific `KtEnumEntry` nodes. Meanwhile the `enum class` itself is represented as a normal `KtClass` node.

The `enum class` itself inherits from `java.lang.Enum`. On the other hand, entries of the enumeration are represented as private classes which inherit from their enum parent. Their constructor therefore calls the one of the `enum class` itself. All classes representing the enum entries are generated thanks to the lowering step described in section 1.3.

Since the enum class will be accessed statically, utility functions such as `values()` and `valueOf(s: String)`, as well as instances of the class entries, are stored inside a module class following the generation of objects (see section 2.1). These two utility functions are fully generated by the compiler.

Take the code shown in figure 2.20 as an example. This simple code would generate a total of 5 files, the content of which is shown in figure 2.21.

```kotlin
enum class Fruits {
    APPLE, BANANA, PEACH
}
```

Figure 2.20: An example of an enum class

```
// Fruits$APPLE.sjsir, Fruits$BANANA.sjsir and Fruits$PEACH.sjsir
class LFruits$APPLE extends LFruits {
  def init___T__I(_name: T, _ordinal: int) {
    this.LFruits::init___T__I(_name, _ordinal)
  }
}
// Fruits.sjsir
class LFruits extends jl_Enum {
  def init___T__I(_name: T, _ordinal: int) {
    this.jl_Enum::init___T__I(_name, _ordinal)
  }
}
// Fruits$.sjsir
module class LFruits$ extends O {
  // enum fields and acessors are omitted

  def valueOf__T__LFruits(string: T): LFruits = {
    if (string.equals__O__Z("PEACH")) {
      this.PEACH$1
    } /* omitting other else if */
    } else {
      throw new jl_IllegalArgumentException()
        .init___T(("Illegal enum constant Fruits." +[string] string))
    }
  }
  val $VALUES: LFruits[]
  def values__ALFruits(): LFruits[] = {
    this.$VALUES.clone__O().asInstanceOf[LFruits[]]
  }
  def init___() {
    this.O::init___();
    this.APPLE$1 = new LFruits$APPLE().init___T__I("APPLE", 0);
    // omitting other entries
    this.$VALUES = LFruits[](this.APPLE$1, this.BANANA$1, this.PEACH$1)
  }
}
```

Figure 2.21: The resulting SJSIR files of the translation of the enum shown in figure 2.20

Enum classes can define a constructor and implement interfaces. This feature is however not yet supported.

2.9.1 Related files

The main source files involved in the translation of interfaces are the following.

- `translate/GenClass.scala` (look for `treeEnumCompanion` and the `genPrimaryConstructor` methods)
- `translate/GenClassUtils.scala` (look for `getEnumDefinitions`, `getEnumCompanionConstructor`) and `KtEnumEntry`
- `utils/Utils.scala` (look for the `enum` keyword)

Test files The tests verifying that interfaces are implemented as expected are found in the following files.

- `classes/MyEnumClass.kt`
- `classes/TestClasses.kt`

2.10 Anonymous objects

Anonymous objects are the Kotlin way of instantiating objects of anonymous classes. Since these are represented as `KtObjectLiteralExpression` in the Kotlin AST, they aren't handled in the lowering phase (see section 1.3).

The way these expressions are handled by the compiler is by generating a class located in a package named after the enclosing declaration of the object. For example, if an anonymous object is declared inside a function `foo()` in a file `Main.kt` then a file `foo/Main$NoNameProvided.sjsir` will be generated.

Figure 2.22 shows how one can declare such an object. After generating the file with the object declaration, the compiler will replace the declaration with an instance of the newly generated class. For the previous example the Scala.js IR would look like shown in figure 2.23.

```kotlin
fun foo() {
  val adHoc = object {
    var x: Int = 0
    var y: Int = 0
  }
  print(adHoc.x + adHoc.y) // prints 0
}
```

Figure 2.22: An anonymous object in Kotlin (from the Kotlin documentation)

```
def foo__V() {
  val adHoc = new Lfoo_Main$NoNameProvided().init___();
  mod:s_Predef$.print__O__V((adHoc.x__I() +[int] adHoc.y__I()));
  (void 0)
}
```

Figure 2.23: The IR representation of code provided in figure 2.22

2.10.1 Related files

The main source files involved in the translation of interfaces are the following.

- `translate/GenExpr.scala` (look for `KtObjectLiteralExpression`)
- `translate/GenClass.scala`
- `translate/GenClassUtils.scala`
- `utils/Utils.scala`

Test files The tests verifying that interfaces are implemented as expected are found in the following files.

- `objects/AnonymousObjects.kt`
- `objects/TestObjects.kt`

2.11 Kotlin js() function

In order to compile the standard library, some way of supporting the special `js(jsCode: String)` function is required. What `js()` does in Kotlin JS is to inline the JavaScript code passed as argument in the final JavaScript file generated by the compiler.

Because the Scala.js IR doesn't offer any way of inlining raw JavaScript, the various calls in the standard library must be treated case by case. This is made possible by the requirement of the argument to the function being a string literal.

This is what the `GenJsFunc` class does. It will extract the argument and generate the corresponding, manually defined, SJSIR nodes.

Take for example the simple call `js("Kotlin.identityHashCode")(3)`. The SJSIR translation of this call is shown in figure 2.24.

```
mod:jl_System$.identityHashCode__O__I(3);
```

Figure 2.24: The translation of a call to the Kotlin `js` function

2.11.1 Related files

The main source files involved in the translation of the Kotlin js function are the following.

- `translate/GenCall.scala` (look for `GenJsFunc` usage)
- `translate/GenJsFunc.scala`

Test files The test file verifying that the specific use cases of the js() function behaves as expected is the following.

- `TestJsFunc.kt`

2.12 Null safety operators

The safe call operator `?.`, the not-null assertion operator `!!` and the elvis operator `?:` were introduced during this project. This required a refined mapping between Kotlin and Scala.js types. In short, Kotlin nullable primitive types are now mapped to their matching Java boxed type. All mappings can be found in the `utils/Utils.scala` source file.

Safe call operator `?.` When encountering safe calls, the Kotlin compiler wraps them inside JavaScript conditional operators. In the case of the Kotlin to Scala.js compiler, this wrapping is translated using a temporary variable and `if` statements. An example is presented in figure 2.25 with its corresponding sjsir representation in figure 2.26.

```
fun dummy() {
  var a: String? = null
  a?.toString()
}
```

Figure 2.25: A safe call on a string value

```
def dummy__V() {
  var a: T = null;
  val tmp$1: T = a;
  if ((tmp$1 !== null)) {
    tmp$1.length__I()
  } else {
    null
  }
}
```

Figure 2.26: The safe call operator translation matching example 2.25

Not-null assertion operator `!!` This operator is translated by the compiler as a not-null assertion. This means that a null check is performed in the same way as for the safe call operator. However, if it is used on a null expression then a `NullPointerException` will be thrown. This is illustrated with an example in figure 2.27 and its corresponding SJSIR equivalent in figure 2.28.

```
fun dummy() {
    var a: String? = null
    a!!
}
```

Figure 2.27: A not-null assertion on a string value

```
def dummy__V() {
  var a: T = null;
  val tmp$1: T = a;
  if ((tmp$1 !== null)) {
    tmp$1
  } else {
    throw new jl_NullPointerException().init___()
  }
}
```

Figure 2.28: The not-null assertion operator translation matching example 2.27

Elvis operator `?:` This operator follows the same logic as previous operators. It allows to return a default value if the expression on its left hand side evaluates to null. An example and its resulting SJSIR translation are provided in figures 2.29 and 2.30.

```
fun dummy() {
  var a: String? = null
  a ?: "Hello World"
}
```

Figure 2.29: Providing a default value on a string value

```
def dummy__V() {
  var a: T = null;
  val tmp$1: T = a;
  if ((tmp$1 !== null)) {
    tmp$1
  } else {
    ("" +[string] "Hello World")
  }
}
```

Figure 2.30: The elvis operator translation matching example 2.29

2.12.1 Related files

The files related to the implementation of these three operators are the following.

- `translate/GenExpr.scala` (look for `KtSafeQualifiedExpression` in the match)
- `translate/GenExprUtils.scala` (look for the `isSafe` variable)
- `translate/GenBinary.scala` (look for the `translateElvis` method)
- `translate/GenUnary.scala` (look for the `EXCLEXCL` token)

Test files The test file verifying that the behavior of these operators works as expected is as follows.

- `TestNullable.kt`

Chapter 3

Gradle tooling

3.1 About Gradle

Gradle [10] is a build tool allowing developers to define the build logic of their project programmatically. This logic can be separated in various tasks performing independent operations. The defined tasks can later be linked among them by defining dependencies in a build script.

The build scripts defining this logic are written in a special DSL (Domain Specific Language) format and since not too long ago in a Kotlin-based DSL. This is probably not by chance since this tool is used by a majority of Kotlin developers (mostly Android) and the Kotlin team itself in the compiler source code.

Besides writing build scripts, developers can write custom plugins [16] which can be applied to existing projects in order to provide reusable logic.

3.2 Compiler plugin

Kotlin JS provides its own Gradle plugin called `kotlin2js` which can be applied in a Gradle project by defining a few dependencies and applying the plugin itself with `apply plugin: 'kotlin2js'`.

The goal for the Kotlin to Scala.js compiler is to provide a similar way of applying our compiler with minor differences with the original one. The source code of the plugin can be found on GitHub [11] along with a simple example project.

3.2.1 Plugin architecture

The plugin is built thanks to the Java Gradle Development Plugin [16] developped by the Gradle team. It provides some useful utilities in order to declare a Gradle plugin and use it in other projects. The original `kotlin2js` plugin is also used in order to provide support for Kotlin SourceSets and allow building a project with the two compilers (see under ”Available tasks”).

Because there are limited examples and documentation about plugin development, all sources are written in Groovy [13]. The base of the plugin is defined inside the `K2SJSCompilerPlugin` file, this is where the compilation task is created and where it's made possible to call it using `gradle build` (see section 3.2.2 for more details).

The compilation of Kotlin sources is achieved by reproducing the compilation pipeline described in section 1.3. In other words, Gradle will invoke the Kotlin to Scala.js compiler with the referenced sources and then hand the generated `.sjsir` files to the Scala.js linker for further treatment.

3.2.2 Plugin usage

Applying the plugin

The basic content of the `build.gradle` file of a Kotlin to Scala.js project is presented in figure 3.1.

Instructions on lines 14 and 15 will add the two dependencies required for the plugin to work, mainly the `kotlin-gradle-plugin` and the `kotlin2sjs` plugin itself.

In order to load the Scala.js library at compile time, it is necessary that the `dependencies` block contains a reference to the right library version (see line 27). Since this is an external dependency, the `repositories` block must be defined as well.

If you wish to compile your code using the original Kotlin compiler, you need to add the corresponding dependencies to the build script. Please refer to the “Avaiable tasks” section below, to learn more on how to run the Kotlin compiler directly.

```
group "ch.epfl.k2sjsir.example"

version "1.0-SNAPSHOT"

buildscript {
  ext.kotlin_version = "1.1.61"

  repositories {
    mavenCentral()
    mavenLocal()
  }

  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    classpath "ch.epfl.k2sjs:kotlin2sjs:0.1-SNAPSHOT"
  }
}

apply plugin: "kotlin2sjs"


repositories {
  mavenCentral()
}

dependencies {
  compile "org.scala-js:scalajs-library_2.12:1.0.0-M2"
  // If you want to make use of the original compiler as well
  //compile "org.jetbrains.kotlin:kotlin-stdlib-js:$kotlin_version"
}
```

Figure 3.1: A typical build.gradle file for a Kotlin to Scala.js project

Available tasks

This section lists the main available tasks defined by the plugin. A full list of these tasks and the ones generated by Gradle can be obtained by running `gradle tasks` in the project root folder.

build This is the standard Gradle build task which builds the sources of project. It depends on the `k2sjs` task defined below.

build-original This task provides an easy way to call the original Kotlin compiler in case one needs to compile a project with both compilers (for benchmarking purposes for instance).

clean This is the standard Gradle clean task. It will erase the content of the `build/` folder generated by the `build` command.

k2sjs This is the main compilation task defining the compilation steps detailed previously.

Available options

Various options are available in order to adapt the `kotlin2sjs` plugin to one's needs. They must be declared and used inside the `k2sjs` configuration block. An example configuration is provided in figure 3.2.

```
// Inside the build.gradle file
k2sjs {
  kotlinHome = "/usr/share/kotlin"
  dstFile = "./web/js/mycode.js"
  compilerOptions = "-verbose"
  optimize = "fullOpt"
}
```

Figure 3.2: An example configuration for the `kotlin2sjs` plugin

kotlinHome Allows to specify the path to the local Kotlin installation directory. This option defaults either to the content of the environment variable `$KOTLIN_HOME` or to `/usr/share/kotlin`.

dstFile This option contains the path and name of the .js file output by the linker. This must match the pattern `path/to/output.js`. This option defaults to `path/to/projectname/build/projectname.js`.

optimize This option defines the optimization level desired for the output JavaScript code. It can contain one of the three following values : ”noOpt”, ”fastOpt” or ”fullOpt”. Its default value is ”fastOpt”.

compilerOptions Allows to specify custom compiler arguments. Those arguments must be contained in a String and be separated by spaces. By default it is empty.

linkerOptions Just like the `compilerOptions`, this field allows to pass custom command line arguments to the Scala.js linker. They must be separated by spaces as well. By default it is empty. Note that the option “-c” to check the consistency of the SJSIR files is always enabled.

Chapter 4

Benchmarks

4.1 Testing environment

In order to verify the efficiency of the translation and compare performance between JavaScript code output by the original Kotlin JS compiler and the Kotlin to Scala.js compiler, different Kotlin benchmarks were implemented.

The benchmarks implemented for this evaluation are the following :

- **DeltaBlue benchmark** This is an implementation of the incremental constraint solver algorithm of the same name. It will mainly benchmark the oriented object capabilities of the compiler.
- **Long Micro benchmark** This is a micro benchmark running binary operations on Longs.
- **Richards benchmark** This is a simulation of an operating system scheduler. It is mainly designed to test operations on arrays.
- **SHA512 benchmark** This is an implementation of SHA512 using Longs.

The Kotlin to Scala.js compiler was used with the `fullOpt` option set on the Scala.js linker. All optimizations available were therefore applied except for the Google Closure Compiler which was disabled. On the other hand, the Kotlin JS compiler was used with no particular options.

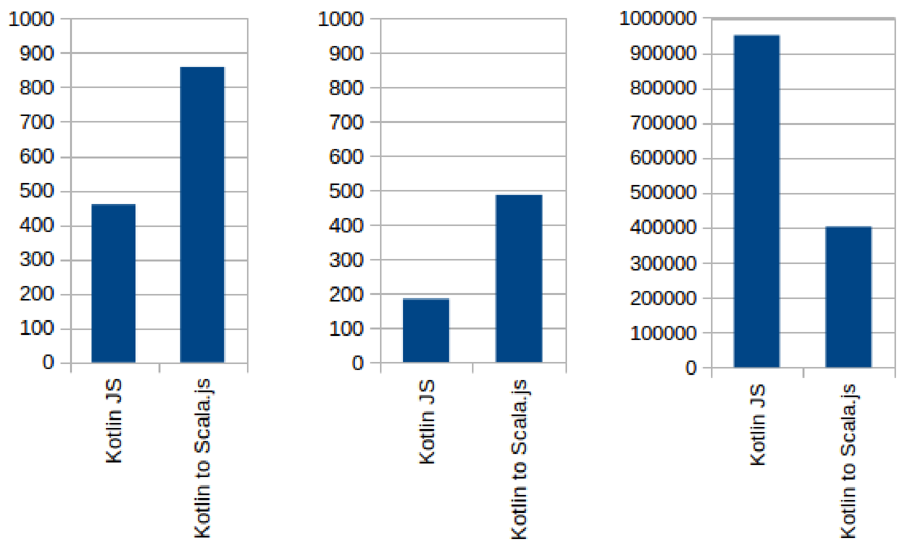
Because the Kotlin standard library is not available yet, a simple LinkedList implementation was used to provide support for lists on both compilers. All benchmarks can be found inside the `test/src/resources/src/benchmarks` folder of the GitHub repository [29].

4.2 Results and analysis

The following results were obtained by compiling the benchmarks with both the Kotlin JS compiler and Kotlin to Scala.js. They were executed in a Linux Mint environment on two web browsers, namely Firefox 57.0.4 and Google Chrome 63.0.3239.132. Both browsers are 64-bit builds.

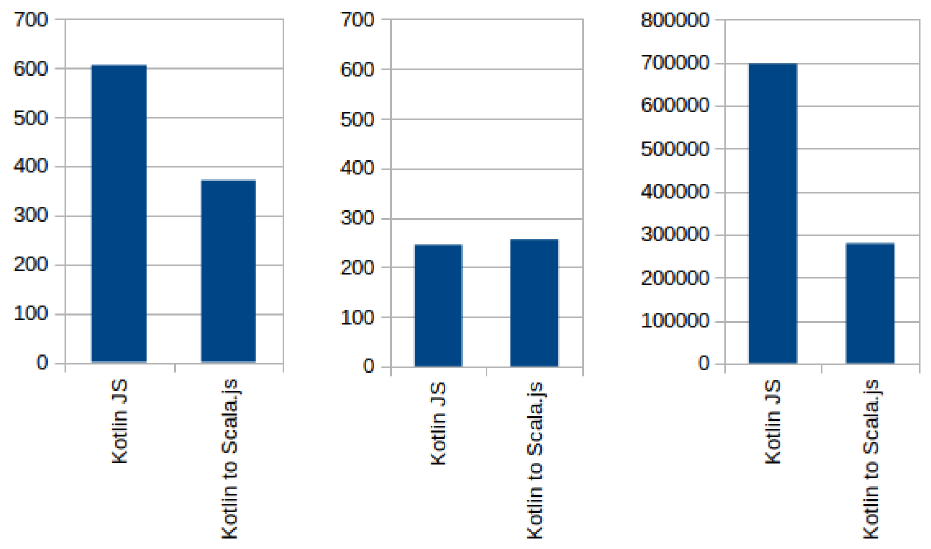
Results for the LongMicro benchmark are presented after substraction of the LongNop result value. This means that the overhead time of computing anything else than an operation on Long values is not taken into account in the results.

The raw data is presented in tables 4.1 and 4.2. In order to better visualize the differences between the two compilers, data for DeltaBlue, Richards and SHA512 benchmarks were grouped in bar charts shown in figures 4.1 and 4.2.



(a) DeltaBlue (b) Richards (c) SHA512

Figure 4.1: Results in Firefox (lower is better)



(a) DeltaBlue (b) Richards (c) SHA512

Figure 4.2: Results in Google Chrome (lower is better)

The results for the DeltaBlue benchmark are interesting. It seems that the implementation of the JavaScript runtime in both browsers differs at some point, leading Kotlin JS to perform better in Firefox whereas the Kotlin to Scala.js compiler has much better results in Google Chrome. It is hard to conclude which of the two compilers performed better here.

Regarding the Richards benchmark, we can see that in Firefox Kotlin JS does better. This is probably because of the difference of translation of arrays between the two compilers. Scala.js keeps close to the JVM implementation of arrays and wraps all of them inside JavaScript classes. This requires a function call before obtaining the array itself and introduces execution overheads. This behavior allows to keep typing informations in the JavaScript code. Kotlin doesn't do that and therefore wins thanks to an implementation of arrays closer to JavaScript semantics.

On the other hand, the execution times in Google Chrome are much closer. It is likely that this is of the same origin as for the DeltaBlue benchmark because of the indirection on array accesses.

Both LongMicro and SHA512 benchmarks perform better with the Kotlin to Scala.js compiler in both web browsers. This large difference in performance

	Kotlin JS		Kotlin to Scala.js	
Benchmark	Time (μs)	SEM	Time (μs)	SEM
DeltaBlue	459.289 10	2.218 71	859.117 47	3.208 04
LongMicro				
(LongNop)	125.476 39	0.701 88	11.388 03	0.021 59
LongXor	21.112 97	0.770 03	162.370 52	0.566 07
LongAdd	165.336 56	1.345 34	198.325 48	0.777 13
LongMul	2561.957 39	4.752 00	204.078 14	0.634 72
LongDiv32_32	6073.693 72	28.940 78	466.435 48	1.080 56
LongDiv32_8	8963.978 14	14.922 87	463.960 42	1.006 94
LongDiv53_53	5985.073 49	26.344 28	596.052 68	1.273 55
LongDiv53_8	9156.009 66	72.357 38	614.255 87	1.599 74
LongDiv64_Pow2	9429.555 44	18.069 75	575.888 60	1.014 34
LongDiv64_64	5813.333 12	13.790 45	1028.320 69	1.733 38
LongDiv64_8	17 033.259 23	21.180 46	1424.533 45	2.370 06
LongToString32	17 519.935 36	151.229 58	552.063 08	1.356 33
LongToString53	28 489.952 17	250.541 57	756.905 14	5.215 76
LongToString64	40 159.856 93	351.844 62	3999.467 57	33.080 56
Richards	184.119 63	0.308 23	487.244 35	1.097 96
SHA512	950 152.000 00	4554.938 41	402 408	8219.831 14

Table 4.1: Results obtained in Firefox for both compilers

is due to the optimized implementation of Longs in the Scala.js library and optimizer.

	Kotlin JS		Kotlin to Scala.js	
Benchmark	Time (μs)	SEM	Time (μs)	SEM
DeltaBlue	606.055 24	1.780 70	370.940 92	1.520 64
LongMicro				
(LongNop)	95.098 02	0.479 85	43.688 18	0.038 76
LongXor	154.294 54	1.605 20	132.636 78	1.044 05
LongAdd	127.472 45	1.588 66	230.626 37	2.325 70
LongMul	1061.115 79	7.456 46	228.638 78	2.218 89
LongDiv32_32	2808.604 78	16.707 20	462.995 40	3.339 24
LongDiv32_8	4144.767 79	29.603 27	445.635 78	3.152 53
LongDiv53_53	2704.164 34	19.209 49	513.256 10	3.333 51
LongDiv53_8	4481.030 02	41.386 67	499.073 08	3.494 03
LongDiv64_Pow2	5165.945 83	42.553 26	490.208 81	3.790 47
LongDiv64_64	2804.607 00	18.105 13	849.861 73	5.544 24
LongDiv64_8	8631.079 29	73.323 33	1366.565 81	6.298 74
LongToString32	8129.340 32	47.134 48	504.820 60	4.379 17
LongToString53	13 103.235 30	93.323 71	685.873 53	6.509 72
LongToString64	17 988.757 39	128.042 66	1680.242 84	8.845 13
Richards	245.065 67	0.494 29	255.698 25	0.520 48
SHA512	696 866.000 00	4694.321 14	279 157.5	11 230.014 27

Table 4.2: Results obtained in Google Chrome for both compilers

Chapter 5

Conclusion

5.1 Project conclusion

This project was (at first) aimed to finish the compilation of the Kotlin JS standard library in order to provide full support of the Kotlin features with the Kotlin to Scala.js compiler. Sadly, more features than expected were missing and a few still are, preventing a complete compilation. The range of unsupported language features has however been greatly reduced and one should easily be able to implement the last missing features.

The complete compilation of the standard library is slowed down as well by the need to write translation for uses of the Kotlin `js` function.

Prior work on the compiler allowed to show that the use of the Scala.js linker for compiling Kotlin to JavaScript would lead to a huge improvement in terms of the size of the final JavaScript code emitted. This project shows that the performance of the Kotlin to Scala.js compiler still has room for improvement in some cases, mainly in the handling of array accesses. As for the other benchmarking results, this compiler is at least as good as what the Kotlin JS compiler would do and even much better in handling 64 bits integer operations.

The current state of the compiler is not suited for large scale projects but could however already be used for smaller sized applications. The fact that the Gradle plugin is fairly easy to configure removes the hard work required to satisfy the compilation pipeline since it doesn't require to operate things manually anymore.

5.2 Future work

The current version of the compiler is still missing important features. In order to complete full classes support, data classes and inner classes must be supported. Destructuring declarations are also missing but can be really handy. They allow, for instance, to retrieve in one line the content of a method returning a tuple.

A strong improvement would be to provide a good algorithm for generating fresh names. As of now the utility function used throughout the project uses the generation of a unique identifier with the help of the UUID package from Java. This works but is however not a recommended way of doing this. This problem could be solved by adding the maintenance of a context which could be passed down to lower layers. Kotlin is already providing such a context but the way it is maintained in the Kotlin compiler is quiet unclear.

The main bottleneck for the use of the compiler is the fact that external classes and dynamic types are not well supported. Such an addition would allow for easy interaction with JavaScript existing features and provide access to the HTML DOM of web pages.

References

[1] "Best practices for top-level declarations".
`https://discuss.kotlinlang.org/t/best-practices-for-top-level-declarations/2198`.

[2] "Hands On ScalaJS".
`http://www.lihaoyi.com/hands-on-scala-js/`.

[3] "Kotlin and Android".
`https://developer.android.com/kotlin/index.html`.

[4] "Node JS".
`https://nodejs.org/en/`.

[5] "Scala.js".
`https://www.scala-js.org/`.

[6] "Scala.js repository on GitHub".
`https://github.com/scala-js/scala-js`.

[7] "ScalaTest FunSuite".
`http://www.scalatest.org/getting_started_with_fun_suite`.

[8] "Understanding modules".
`https://discuss.kotlinlang.org/t/understanding-modules/474/2`.

[9] "What's Kotlin Backing Field For?"
`https://stackoverflow.com/questions/43220140/whats-kotlin-backing-field-for`.

[10] Gradle build tool.
`https://gradle.org/`.

[11] Alonso, F. "A Gradle plugin for the Kotlin to Scala.js compiler".
`https://github.com/flonso/kotlin-scalajs-gradle-plugin`.

[12] Breslav, A. “Improving Java Interop: Top-Level Functions and Properties”.
https://blog.jetbrains.com/kotlin/2015/06/improving-java-interop-top-level-functions-and-properties/.

[13] Foundation, T. A. S. “Apache Groovy”.
http://groovy-lang.org/.

[14] Fränkel, N. “Scala vs Kotlin: Pimp my library”.
https://blog.frankel.ch/scala-vs-kotlin/1/#gsc.tab=0.

[15] Gardner, B. “‘introduction to gradle’”.
https://www.bignerdranch.com/blog/introduction-to-gradle/.

[16] Gradle. “Implementing Gradle plugins”.
https://guides.gradle.org/implementing-gradle-plugins/.

[17] Inc., G. Google closure compiler.
https://developers.google.com/closure/compiler/.

[18] Jemerov, D. “Kotlin 1.1.60 is out”.
https://blog.jetbrains.com/kotlin/2017/11/kotlin-1-1-60-is-out/.

[19] JetBrains. “Higher-Order Functions and Lambdas”.
https://kotlinlang.org/docs/reference/lambdas.html.

[20] JetBrains. “JavaScript DCE”.
https://kotlinlang.org/docs/reference/javascript-dce.html.

[21] JetBrains. “Kotlin Blog”.
https://blog.jetbrains.com/kotlin/.

[22] JetBrains. “Kotlin Code Examples”.
https://github.com/JetBrains/kotlin-examples.

[23] JetBrains. “Kotlin Lang”.
https://kotlinlang.org/.

[24] JetBrains. “Kotlin Lang Reference”.
https://kotlinlang.org/docs/reference/.

[25] JetBrains. “Kotlin Language Specification”.
https://jetbrains.github.io/kotlin-spec/.

[26] JetBrains. “Kotlin repository on GitHub”.
https://github.com/JetBrains/kotlin.

[27] JetBrains. “The Gradle plugin for Kotlin”.
https://github.com/JetBrains/kotlin/tree/master/libraries/tools/kotlin-gradle-plugin/src/main/kotlin/org/jetbrains/kotlin/gradle/plugin.

[28] JetBrains. “Type Checks and Casts: ’is’ and ’as’ ”.
https://kotlinlang.org/docs/reference/typecasts.html.

[29] Lionel Fleury, Guillaume Tournigand, F. A. “Kotlin to Scala.js compiler”.
https://github.com/flonso/Kotlin-Scala.js.

[30] Lionel Fleury, G. T. “From Kotlin to Javascript using the Scala.js compiler”.

[31] Lionel Fleury, G. T. “Kotlin to Scala.js compiler”.
https://github.com/lionelfleury/Kotlin-Scala.js.

[32] Martin, D. “Getters and Setters in Scala”.
https://www.dustinmartin.net/getters-and-setters-in-scala/.

[33] Sieling, G. “Scala Lambda Example”.
https://www.garysieling.com/blog/scala-lambda-example.

[34] Tâche, G. “A ScalaJS Gradle plugin”.
https://github.com/gtache/scalajs-gradle.