Production Level Tooling for the Kotlin to Scala.js Compiler

Master Semester Project

Florian Alonso

EPFL - Programming Methods Laboratory (LAMP)

Contents

1. Introduction
2. Main improvements
3. Gradle Plugin
4. Benchmarks & Results
5. Future work

Introduction

Project Goals

- Continue the work started by Guillaume Tournigand and Lionel Fleury
- Validate the generic nature of the Scala.js IR
- Compile Kotlin Standard Library to the Scala.js IR
- Develop Gradle tooling

Main improvements

Main improvements

Binary and unary operations

Binary and unary operations

Endured an important refactoring because of Scala.js internal changes.

- Each IR type has its own representation
- Appropriate conversions were needed

The resulting logic is closer to what is done in Scala.js itself.

Main improvements

Null safety operators

Null safety operators

Kotlin provides its own way of avoiding (or dealing with) NullPointerExceptions :

- Safe calls with `?.`
- Default values with the elvis operator `?:`
- Not-null assertions with `!!`

These operators are translated as conditional expressions.

Expressions to be tested are first assigned to temporary variables.

Null safety operators - Safe call and elvis operator

```kotlin
fun nullSafety(canBeNull: String?): Int {
    return canBeNull?.length ?: 0
}
```

Null safety operators - Safe call and elvis operator

```
fun nullSafety(canBeNull: String?): Int {
    return canBeNull?.length ?: 0
}
```

```
static def nullSafety__T__I(canBeNull: T): int = {
  val tmp$1: T = canBeNull;
  if ((tmp$1 !== null)) {
    tmp$1.length__I()
  } else {
    null
  }
}
```

Null safety operators - Safe call and elvis operator

```kotlin
fun nullSafety(canBeNull: String?): Int {
    return canBeNull?.length ?: 0
}
```

```scala
static def nullSafety__T__I(canBeNull: T): int = {
  return {
    val tmp$0: jl_Integer = {
      val tmp$1: T = canBeNull;
      if ((tmp$1 !== null)) {
        tmp$1.length__I()
      } else {
        null
      }
    };
    if ((tmp$0 !== null)) {
      tmp$0.asInstanceOf[I]
    } else {
      0
    }
  }
}
```

Null safety operators - Not-null assertion

```
fun assertNotNull(cantBeNull: Any?): Unit {
    cantBeNull!!
}
```

Null safety operators - Not-null assertion

```
fun assertNotNull(cantBeNull: Any?): Unit {
    cantBeNull!!
}
```

```
static def assertNotNull__O__V(cantBeNull: any) {
  val tmp$0: any = cantBeNull;
  if ((tmp$0 !== null)) {
    tmp$0
  } else {
    throw new jl_NullPointerException().init___()
  }
}
```

Main improvements

Properties accessors

Properties accessors

In Kotlin, properties can either make use of default accessors or define custom ones.

When a property makes use of a default accessor or uses the `field` keyword, a backing field is generated.

Properties accessors - Backing field in Scala

The `field` keyword is simply a placeholder for the compiler-generated backing field.

```
class Accessors {
  var counter = 0
    get() = field
    set(value: Int) { field = value }
}
```

Properties accessors - Backing field in Scala

The `field` keyword is simply a placeholder for the compiler-generated backing field.

```
class Accessors {
  var counter = 0
    get() = field
    set(value: Int) { field = value }
}
```

In Scala, a backing field would look like this :

```
class Accessors {
  private var _counter = 0 // <- the backing field
  def counter = _counter
  def counter_= (value: Int) = { _counter = value }
}
```

Properties accessors - Kotlin example

```kotlin
class Accessors {
  var counter = 0 // with backing field

  val msg // without backing field
    get() = "Hello Kotlin"
}
```

Properties accessors - Kotlin example

```
class Accessors {
  var counter = 0 // with backing field

  val msg // without backing field
    get() = "Hello Kotlin"
}
```

```
class LAccessors extends O {
  var counter: int
  def counter__I(): int = { this.counter }
  def counter__I__V(set: int) {
    this.counter = set
  }
  def msg__T(): T = {
    ("" +[string] "Hello Kotlin")
  }
  def init___() {
    this.O::init___();
    this.counter = 0
  }
}
```

Main improvements

Interfaces

Interfaces

- Interfaces are translated with abstract method bodies
- The default implementations are stored in a separate class

Interfaces - A default implementation

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

INTERFACES - A DEFAULT IMPLEMENTATION

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

```
interface LDummy {
  def a__T(): T = <abstract>
}

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

Interfaces - Bridge generation

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

class MyClass: Dummy
```

Interfaces - Bridge generation

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

class MyClass: Dummy
```

```
class LMyClass extends O implements LDummy {
  def a__T(): T = {
    LDummy$DefaultImpls::a__LDummy__T(this)
  }
  // init
}
```

Main improvements

Super calls

Super calls

Super calls must also be translated depending on which instance they refer to :

- Reference to a (parent) class
- Reference to an interface's default implementation

SUPER CALLS

Interface :

- `ApplyStatic()` applies a static method for a given class type.

Parent class :

- `ApplyStatically()` indicates a static dispatch to a given class type on the receiver's instance ($\tilde{=}$ invokespecial).

Super calls - Example

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

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

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

  fun example() {
    super<I>.a
    super<MyParent>.a
    this.a
  }
}
```

Super calls - Example

```
class LMyChild extends LMyParent implements LI {
  // 'a' field accessors

  def example__V() {
    LI$DefaultImpls::a__LI__T(this); // ApplyStatic
    this.LMyParent::a__T(); // ApplyStatically
    this.a__T();
    (void 0)
  }

  // init call
}
```

Main improvements

Lambdas

Lambdas

In Kotlin JS, anonymous functions are JavaScript functions.

In Scala.js, they are wrapped and are JavaScript objects.

To stick to the Kotlin JS semantics, translate them as closures :

- Formal parameters of type `Any`
- Inside the body, parameters are cast to the declared type

LAMBDAS - EXAMPLE

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

Lambdas - Example

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

```
// Inside the init call of a class
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)
})
```

Main improvements

Kotlin `js` function

Kotlin `js` function

Allows to write pure JavaScript inside Kotlin code:

```
js("Kotlin.identityHashCode")(3)
```

Breaks the compiler abstraction. The Scala.js IR doesn't allow that.

It must therefore be handled case by case.

Main improvements

Other additions

OTHER ADDITIONS

- Qualified this
- Type checks and casts
- Enum classes
- Anonymous objects

Gradle Plugin

Plugin overview

- Pass the Kotlin source files to the compiler
- Output the SJSIR files
- Use the Scala.js linker to produce a JS file
- Success !

Plugin setup - build.gradle

```
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" // <- the plugin

repositories { mavenCentral() }

dependencies {
  compile "org.scala-js:scalajs-library_2.12:1.0.0-M2"
  compile "org.jetbrains.kotlin:kotlin-stdlib-
  ↪  js:$kotlin_version"
}
```

Plugin setup - options

The plugin offers various options to setup the compiler or the Scala.js linker. An example would be :

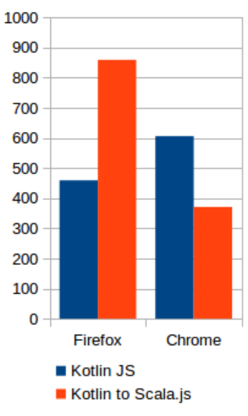
```
// build.gradle
k2sjs {
  optimize = "fullOpt"
}
```

Benchmarks & Results

3 benchmarks were implemented to test the compiler :

- DeltaBlue, testing object oriented capabilities
- Richards, testing array operations
- SHA512, testing `Long` operations

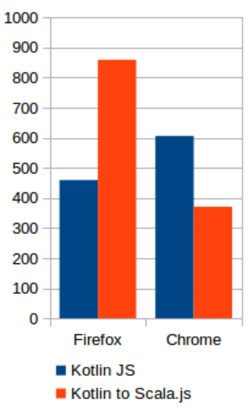
Results



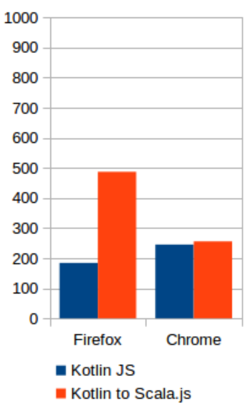
(a) DeltaBlue

Results in Firefox and Chrome (execution times, lower is better)

Results



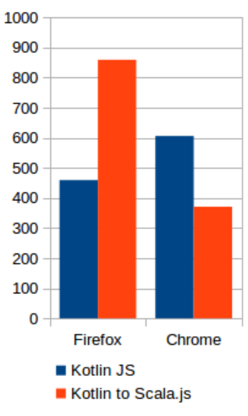
(a) DeltaBlue



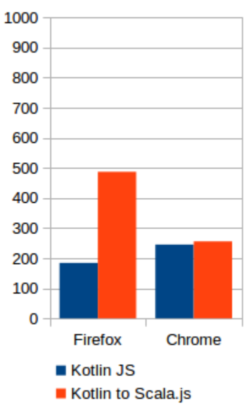
(b) Richards

Results in Firefox and Chrome (execution times, lower is better)

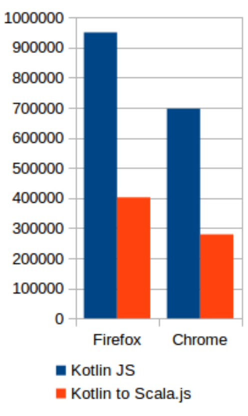
Results



(a) DeltaBlue



(b) Richards



(c) SHA512

Results in Firefox and Chrome (execution times, lower is better)

Future work

Future work

- Suited for small size applications
- Regarding performance both compilers have their flaws
- Kotlin Standard Library is still not fully compilable
- Interactions with HTML DOM are hard, poor dynamic types handling

Questions?

Backup slides

Type checks and casts

- Kotlin provides two keywords in order to perform type checking (with `is`) or casts (with `as`) :

```
myVal is SomeClass
```

Type check

```
myVal as SomeClass
```

Type cast

- The current version of the compiler doesn't support smart casts.

Type checks and casts - a complete example

```
fun casts(anyVal: Any): Unit {
  if (anyVal is Dummy) {
    val d = (anyVal as Dummy)
    // Use d
  } else if (anyVal is Int) {
    val i = anyVal as Int
    // Use i
  } else {
    val s = anyVal.toString()
    // Use s
  }
}
```

Type checks and casts - A complete example

```
static def casts__O__V(anyVal: any) {
  if (anyVal.isInstanceOf[LDummy]) {
    val d: LDummy = anyVal.asInstanceOf[LDummy];
    (void 0)
  } else if (anyVal.isInstanceOf[jl_Integer]) {
    val i: int = anyVal.asInstanceOf[I];
    (void 0)
  } else {
    val s: T = anyVal.toString__T();
    (void 0)
  };
  (void 0)
}
```

Qualified this

The keyword `this` must be translated in two different ways :

- As the SJSIR `This()` object
- As a reference to the `$this` argument (for extension functions for instance)

Enum classes

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

- Enum entries are classes themselves
- The compiler generates the `values()` and `valueOf(s: String)` methods.
- All entries and functions are stored inside a module class

Enum classes - A fruity example

```
// 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)
  }
}
```

Enum classes - Fruits$.sjsir

```
module class LFruits$ extends O {
  def valueOf__T__LFruits(string: T): LFruits = {
    if (string.equals__O__Z("PEACH")) {
      this.PEACH$1
    } else { /* ... */
      throw new jl_IllegalArgumentException()
        .init___T(/* ... */)
    }
  }
  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)
  }
}
```

Anonymous objects

- Translated as normal objects
- Declaration is replaced with a `New()` SJSIR node

ANONYMOUS OBJECTS - EXAMPLE

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

```
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)
}
```