Cross-Platform Language Design

Sébastien Doeraene

EPFL
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

It is better to repent a sin than regret the loss of a pleasure.

— Oscar Wilde

Acknowledgments

Abstract

Programming languages are increasingly compiled to multiple runtimes, each featuring their own rich structures such as their object model. Furthermore, they need to interact with other languages targeting said runtimes. A language targeting only one runtime can be designed to tailor its semantics to those of that runtime, for easy interoperability with other languages. However, in a language targeting multiple runtimes with differing semantics, it is difficult to cater to each of them while retaining a common behavior across runtimes. We call *cross-platform language* a language that aims at being both *portable* across platforms and *interoperable* with each target platform. Portability is the ability for a program or a library to cross-compile for multiple platforms, and behave the same way on all of them. Interoperability is the ability to communicate with other languages on the same platform. While many cross-compiling languages focus on one of these two properties—only adding support for the other one as an afterthought—, languages that are designed from the ground up to support both are rare.

In this thesis, we present the design of Scala.js, the dialect of Scala targeting the JavaScript platform, which turned Scala into a cross-platform language. On the one hand, Scala programs can be cross-compiled for the JVM and JavaScript with portable semantics. On the other hand, whereas Scala/JVM interoperates with Java, Scala.js interoperates with JavaScript, allowing to use any JavaScript library. Along the dissertation, we give insights that can be transferred to the design of other cross-platform languages, although with a bias towards those targeting the JVM and JavaScript.

The first and most obvious challenge is to reconcile the static nature of Scala's object model with JavaScript's dynamic one. Besides the ability to mutate a class hierarchy at run-time in JavaScript, there are fundamental differences between the two models, in particular the difference between compile-time overloading and run-time overloading. We discuss how such semantic mismatches can live in harmony within the language.

The second challenge is to obtain good performance from a language where interoperability with a dynamic and unknown part of the program is pervasive. To that end, we design and specify an intermediate representation (IR) with first-class support for dynamically typed interoperability features in addition to statically typed JVM-style operations. Despite its tight integration with the open world of JavaScript, most of the IR can be considered as a closed

Résumé

De plus en plus fréquemment, les langages de programmation sont compilés vers plusieurs environnements d'exécutions (runtimes), chacun avec leurs riches structures telles que leur modèle objet. En outre, ils se doivent d'interagir avec les autres langages qui y résident. Un langage visant un unique runtime peut tailler ses sémantiques sur mesure pour celui-ci, de sorte à présenter une interopérabilité aisée avec les autres langages. En revanche, dans un langage qui vise plusieurs runtimes avec des sémantiques divergentes, il est difficile de se spécialiser pour chacune d'elles tout en conservant un comportement commun entre les runtimes. Nous appelons *langage transplateforme* un langage qui aspire à être à la fois *portable* à travers plateformes et *interopérable* avec chacune des plateformes visées. La portabilité est la faculté d'un programme ou d'une bibliothèque à compiler vers plusieurs plateformes tout en préservant une sémantique commune. L'interopérabilité est sa capacité à communiquer avec d'autres langages sur la même plateforme. Tandis qu'il existe de nombreux langages qui se focalisent sur l'une ou l'autre de ces deux propriétés — la seconde n'étant ajoutée qu'après coup —, les langages conçus pour le support des deux se font rares.

Dans cette thèse, nous présentons la conception de Scala.js, le dialecte de Scala visant la plateforme JavaScript, qui a fait de Scala un langage transplateforme. D'une part, les programmes Scala peuvent être compilés vers la JVM et vers JavaScript avec une sémantique portable. D'autre part, tandis que Scala/JVM interopère avec Java, Scala.js interopère avec JavaScript, permettant ainsi d'utiliser toute bibliothèque JavaScript existante. Au travers de la thèse, nous donnons des idées pouvant être transférées à la conception d'autres langages transplateformes, avec toutefois un certain biais en faveur de ceux visant la JVM et JavaScript.

Le premier défi, et le plus évident, est de réconcilier la nature statique du modèle objet de Scala avec celle dynamique de JavaScript. Au-delà de la capacité de JavaScript à modifier la hiérarchie de classes à l'exécution, il existe des différences fondamentales entre les deux modèles, en particulier la différence entre la surcharge (overloading) à la compilation et celle à l'exécution. Nous traitons des moyens par lesquels de telles disparités sémantiques peuvent vivre en harmonie au sein du langage.

Le second défi est d'obtenir de bonnes performances malgré l'interopérabilité omniprésente avec une portion du programme dynamique et inconnue. Dans ce but, nous concevons et spécifions une représentation intermédiaire (IR, intermediate representation) intégrant des

Contents

Chapter 1

Introduction

"JavaScript is the Assembly language of the Web." The saying goes back almost a decade, and has become more true than many people would like it to be. Even though the arrival of WebAssembly [29] clearly gives an entirely new dimension to the "Assembly of the Web", it has not yet dethroned the hundreds of languages that can be compiled to JavaScript. From the simple syntactic sugar (CoffeeScript [10]) to the sophisticated type systems (TypeScript [45], Flow [21]) to the purely functional with vastly different run-time semantics (PureScript [53] and Elm [19]) to ports of languages initially designed for other runtimes (GWT [25], ClojureScript [9], FunScript [22]), the language landscape for Web front-end—and other JavaScript-based technologies—has considerably grown.

One aspect where the Assembly analogy breaks down is on *interoperability*. Whereas a language compiling to actual Assembly typically does not need to interoperate with hand-written Assembly code, a language targeting JavaScript can hardly ignore JavaScript libraries entirely. At the very least, manipulating the Document Object Model (DOM)—the model of the visual rendering of a Web page—is necessary for any kind of interaction with the user. While it is possible to constrain every interaction to manipulate a WebGL canvas, it is quite limiting. Moreover, a developer would want to leverage the huge ecosystem of JavaScript libraries.

Interoperability with JavaScript libraries is therefore required, in some form or another. The extent to which languages support interoperability varies, going from the constrained asynchronous message passing of Elm to the deep object-oriented interactions in ClojureScript. This raises the question: what is enough? One objective measure we can use, which is the driving factor in Section 2.2, is whether we can use all existing JavaScript libraries, or whether some of them are beyond the reach of the language, requiring bridges written in hand-written JavaScript.

Besides interoperability with JavaScript libraries, ports of languages initially targeting different runtimes, as well as languages specifically designed to target multiple runtimes, face the challenge of *portability*. In such languages, there typically exists a more or less large subset

of the language that can be compiled to multiple runtimes, with more or less equivalent semantics. Here also, the extent of this subset varies across languages, ranging from virtually nonexistent (usually by design, as in PureScript if we compare it to its big brother Haskell, in this case due to the strict versus lazy evaluation semantics) to mostly equivalent with exceptions (for example, numbers in ClojureScript versus Clojure) to near-perfect emulation (typically found in embedded VMs such as Doppio [66]).

The quality of a language's portability has a direct consequence on its ecosystem of libraries. Specifically, how many libraries can be reused across runtimes, and how easily. In practice, even if the language itself is portable, unless the ecosystem follows suit and publishes portable libraries, it remains difficult to do any meaningful job spanning several runtimes. An example of this phenomenon, as of 2018, is Kotlin [33], which is itself relatively portable, but whose ecosystem, for the most part, sticks to the JVM target. A plausible cause is that until recently (November 2017), Kotlin did not provide a good practical story for a library to target multiple platforms. At the other end of the spectrum, Haxe [30], whose top-level documentation pages include a showcase of cross-platform libraries, is probably the language most known for having a multi-runtime ecosystem.

These two properties, interoperability and portability, are at the heart of what we call a cross-platform language. Libraries and applications can *cross*-compile for multiple targets and behave the same way everywhere, and they can interoperate with their respective *platforms*. Surprisingly, few languages are really designed from the get-go to extensively feature both properties at once, focusing instead on one of them and patching some support for the other as an afterthought. This thesis aims at tackling this dual goal in a principled way, essentially doing cross-platform language design.

Although the "philosophy" described in this thesis should transfer to any cross-platform language, we will focus our technical discussion on the challenges and decisions arising from combining JavaScript and the JVM as two target platforms. More specifically, we study the design of Scala.js [16], the dialect of Scala targeting the JavaScript platform. This dialect was designed and implemented from the very early stages (literally the first few weeks) with the goal of interoperability in mind. Very quickly, we realized that the dialect could not take off without strong portability as well, due to the existing ecosystem of libraries targeting the JVM and relying on more JVM-specific behaviors than one would like. This pair of goals turned out to be somewhat conflicting, as we will see in Chapter 2, forcing the design to include some trade-offs on one side or the other. On top of that, since Scala.js was meant to be a practical tool for production use, performance of the compiler and the generated code was an important concern, and that, too, forced some trade-offs with interoperability and portability. This dissertation articulates those three goals throughout.

1.1 Software

Although this dissertation does not directly depend on it, we cannot ignore the actual implementation of Scala.js. Its main website is https://www.scala-js.org/, and its source code, at the time of writing, is available on GitHub at https://github.com/scala-js/scala-js.[1] More specifically, this thesis refers to Scala.js 1.0.0-M5. The file `DEVELOPING.md` gives information about the structure of the project, and how to build and test it. Besides the author, the main contributors to this project, and more importantly the ideas and designs behind it, are Tobias Schlatter and Nicolas Stucki.

1.2 Overview

This dissertation comprises 3 chapters, going from the designs at the Scala.js source code level, down to the implementation.

Chapter 2 gives further motivation for portability and interoperability, and makes those notions more precise. We describe the main portability challenges of Scala.js and how they are overcome. We briefly evaluate portability in terms of how much of the Scala/JVM test suites pass on Scala.js. Afterward, we dive deeper into the design of interoperability between Scala.js and JavaScript. We give an objective measure of interoperability, which is completeness with respect to the semantics of the host language, and show that Scala.js satisfies this measure—with the exception of one particular JavaScript feature, namely `new.target`.

Because Scala as a source language is a complex beast, reasoning well enough about it to optimize it is virtually impossible. Chapter 3 gives a specification of the Intermediate Representation (IR) used by Scala.js, which tightly integrates portability and interoperability features. We also present some fundamental properties of the IR that are necessary to build optimizations on top of it.

Finally, Chapter 4 discusses how the IR is optimized and eventually compiled down to JavaScript. In particular, we elaborate on the incremental parallel nature of the whole-program optimizer and on specific optimizations that we apply to 64-bit integers. We conclude with a performance evaluation of programs emitted by the Scala.js compiler toolchain.

---

[1]Should this link be invalidated in the future, it should always be possible to find the source code through the main website.

1.3 Contributions

The main contributions of this thesis are the following:

- We give an objective measure for the quality of interoperability in a cross-platform language, as completeness with respect to the semantics of the host language. The criterium is given in Section 2.2.6, and drives the design of all the interoperability features of Scala.js in Section 2.2. It can be used in the design of other cross-platform languages, irrespective of their target platform. Completeness gives the guarantee that programs written in the cross-platform language can use any library of the host language.

- We give the definition of an intermediate representation, the Scala.js IR, with precise syntax, type system and run-time semantics, in Chapter 3. This IR is used as a contract between the compiler front-ends (for various versions of Scala.js, or even other languages) and the optimizer and emitter to JavaScript. It features both statically typed JVM-like operations and dynamically typed JavaScript operations in a unified type system. The type system is expected to be sound, and IR programs live in a closed world, which provides the foundations for whole-program optimizations to be applied.

- We show a novel approach to designing parallel and incremental whole-program optimizers, in Section 4.2. The approach is based on so-called knowledge queries, which create a *modular* interface between incremental changes in a program and the optimizations they invalidate. We used this approach to build the whole-program optimizer of Scala.js, whose incremental runs exhibit 10 to 100 running time improvements over batch runs.

- We present a fast implementation of 64-bit integers for the JavaScript platform, in Section 4.4. It is 3.6x to 60x faster than existing approaches, and only about 3–5x slower than the native implementation of 32-bit integers. This implementation could transfer to any statically typed language compiling to JavaScript, solving the tension found in previous compilers to JavaScript between fast-but-incorrect and correct-but-slow 64-bit integers.

Chapter 2

Cross-Platform Language Semantics

At the heart of every programming language stands its semantics. In the context of a cross-platform language, the semantics needs to address both portability and interoperability. As we introduced in the previous chapter, portability is the property that a program can compile and have the same behavior across platforms, i.e., its semantics is portable across platforms. Interoperability is the ability to communicate with other languages on each target platform, i.e., the semantics needs to be shared with other languages on the same platform. Obviously, not all of a language's semantics can be portable and interoperable at the same time. If that were true, the various platforms would have equivalent semantics, and we would therefore not consider them as separate platforms to begin with. Nevertheless, featuring both portability and interoperability is not a lost cause. Rather, some aspects of the language semantics should target portability, and some others should target interoperability.

In the context of Scala.js, portability is measured with respect to Scala/JVM, while interoperability is measured with respect to JavaScript. In this chapter, we explore the most relevant semantics of Scala.js as seen from those two angles. We first build the foundations through portability, then expand the reach of Scala.js through interoperability. This makes sense because of the prior existence of Scala/JVM, which constrains the design. When designing a fresh language, those two processes are much more interleaved, and woven together. Despite the constraints imposed by Scala/JVM, some portability was sacrificed after the fact in Scala.js to improve interoperability and performance, and will be described in the respective sections.

2.1   Building the Foundations: Portability

Since the JVM has long been the only platform supported by Scala,[1] there are two different ways one can think about the semantics of Scala. On the one hand, we can follow the Scala

---

[1]An effort to port Scala to .NET was attempted but later abandoned.

Language Specification [48] to the letter. On the other hand, we can observe how Scala/JVM behaves and define the semantics of Scala as the observed behavior. In theory, the former is the obvious good choice, because it is a principled approach, and because the specification leaves the semantics of some constructs as implementation-defined, potentially increasing our freedom as the designer of a dialect for another platform. Unfortunately, in practice we must account for an external agent: the ecosystem of existing libraries. As we saw in the previous chapter, when designing a cross-platform language, it is not sufficient for the language itself to be portable; so must the libraries. The trouble is that, being exposed only to Scala/JVM, library developers have (perhaps unknowingly) relied on some behavior of Scala/JVM that is technically implementation-defined according to the specification. These behaviors are de facto semantics, and must be taken into account when considering portability of the language. As our implementation of Scala.js reached more libraries, we progressively realized that virtually every corner of the specification that was left to the implementation had been turned into de facto semantics in more or less critical ways. Therefore, we had no choice but to consider de facto semantics as an integral part of the Scala specification, and we sometimes call the combination of both as the Scala/JVM specification.

In this section, we discuss some parts of the Scala/JVM specification, including de facto semantics, that have a non-trivial impact on Scala.js, notably on interoperability. Beyond the case of Scala.js, illustrating these aspects can also serve as a check-list of things to look out for when designing a cross-platform language from scratch. Throughout this section, we justify decisions based on arguments such as "the ecosystem did/did not rely on some specific behavior". Those observations came from experience, trying to implement alternative behaviors and watching the ecosystem survive them, or breaking down. We did not actually record empirical data as we progressed, therefore those arguments unfortunately remain anectodal.

The "ecosystem" itself is difficult to define. For Scala.js, it even progressed over time, as the development of the language matured. At first, approximately until Scala.js 0.4.x, i.e., one year of development, it only consisted of early adopters: users who were willing to play with an immature technology and report immediate limitations. The feedback from early users was crucial to make the language usable at all. Once Scala.js addressed immediate needs, the pool of early adopters grew into a small community of enthusiasts. That community developed initial libraries, or tried to port existing libraries, exhibiting new requirements. So far, only users specifically interested in the JavaScript platform tried to use Scala.js. The last critical step was to make the language portable enough for large existing libraries to cross-compile as-is, and to provide the necessary tooling to make it as easy as possible to do so, which was the main focus of the transition from 0.5.x to 0.6.x, approximately two years into the development. At that point, the ecosystem suddenly exploded, as more and more established libraries from the Scala/JVM ecosystem started cross-compiling for Scala.js. In the process, the test suites of these libraries started exercising all the possible obscure de facto semantics of Scala/JVM. Unlike the two first phases of the development, where requirements and bugs were only elicited by isolated experiments done by users, this third phase creates many more

test cases for the language without any user involvement.

2.1.1 Primitive numeric types

Scala and the JVM have several primitive numeric types: `Byte`, `Short`, `Int`, `Long`, `Float` and `Double`. The first four are signed fixed-width integer types, while the last two are IEEE 754 floating point number types. JavaScript, on the other hand, only has one primitive numeric type, called `number`, which is a 64-bit floating point number types (coinciding with Scala's `Double`).

When cross-compiling a language to multiple platforms, it is tempting to adapt the semantics of primitive numeric types to that of the target platform. After all, this makes for an easy implementation, and—supposedly—better performance for primitive operations. If we do this, however, an addition on `Int`s will not behave the same way on JavaScript as on the JVM. For example, `1000000000 + 2000000000` will give `3000000000` on JavaScript but `-1294967296` (due to arithmetic modulo 2^{32}). This is a serious threat to portability, which must be avoided.

Correctly implementing Scala-style arithmetics on JavaScript turns out to be fairly easy, thanks to the `asm.js` encoding [3]: for example, for `Int`s, `a + b` in Scala would be translated to `(a + b) | 0` in JavaScript, maintaining the proper semantics. There is however a major difficulty for `Long`s, whose 64 bits worth of precision cannot so much as be *stored* in a JavaScript `number`. Many languages cross-compiling to JavaScript abandon correctness for their 64-bit integers (using `number`s instead, hence reducing the precision), which causes them not to be portable, while the others implement them correctly but with excessive performance overheads. Efficiently implementing 64-bit integers on JavaScript is a hard problem. We will discuss how we solved that issue in details in Section 4.4.

2.1.2 Boxed classes for primitive types

In Java, primitive types cannot be used in generic contexts. For example, one cannot declare a `List<int>`. Instead, we must use a `List<Integer>`, where `Integer` is the *boxed class* for `int`. A boxed class is a tiny immutable wrapper over a primitive value. The Scala language relaxes this limitation, and allows the use of primitive types in generic contexts. However, when we manipulate a `List[Int]`, what actually happens behind the scenes is that the compiler rewrites it as a `List[Integer]`, and adds all the necessary instructions to box and unbox the primitive values (along with more advanced magic for overriding bridges).

Although the Scala Specification is silent about this phenomenon, it can be observed:

```
scala> def test(x: Any): String = x match {
     |     case x: java.lang.Integer => "an Integer"
     |     case x: Int               => "an Int"
     | }
```

```
test: (x: Any)String

scala> test(5)
res0: String = an Integer
```

It turns out that the ecosystem of Scala libraries (including the standard library) relies on being able to match primitive values as if they were instances of their corresponding boxed class, and vice versa (match instances of boxed classes as if they were primitive values).

When designing the semantics of Scala.js, it is therefore—perhaps unfortunately—necessary to preserve this behavior. While it is easy to preserve it as far as portability is concerned, it is a threat to interoperability: suppose we have a type `js.Array[T]` representing JavaScript arrays; a `js.Array[Double]` should naturally represent the type of a JavaScript array of numbers, but in fact it represents the type of a JavaScript array of `java.lang.Double`s! We will see in Section 2.2 how we can design the interoperability semantics in a way that avoids this issue.

For now, the easiest way to explain what we do is that, unlike on the JVM, primitives are *not* boxed when they enter generic contexts (including being upcast to `Any`), i.e., they are not stored in actual instances of a class, such as `java.lang.Integer`. Instead, an `Int` stored in an `Any` remains as a primitive **number** value in JavaScript. This is possible because JavaScript is dynamically typed, so any kind of value can be stored in any variable.[2] The above deviation from the Scala semantics allows to improve interoperability. For example, a `js.Array[Int]` effectively contains primitive **number**s. However, since we argued that type-testing such a "non-boxed" `Int` against a `java.lang.Integer` must succeed, we must also specify that type-testing a primitive **number** against a `java.lang.Integer` succeeds, at least if its value fits in the range of an `Int`. We generalize this for all primitive numeric types except `Long`s, with their respective ranges. This has one unwanted consequence: that a value which started its life as an `Int` can later match a type-test against a `Double`, and vice versa (for `Double` values that do fit in an `Int`):

```
 test(5.0) // prints "an Integer" as well!
```

This behavior is different than on the JVM, so this is a threat to portability. However, the benefits in terms of interoperability were too great to ignore. Moreover, it turned out that the Scala ecosystem was not relying on this specific behavior of type-tests on primitives (unlike the unboxed versus boxed type tests). If the ecosystem had been broken by this change, we would not have been able to apply this tweak to the semantics, and interoperability would have suffered.

There is another subtle consequence of the same change: the result of `toString()` is slightly altered for `Double`s whose value happens to be whole. In Scala/JVM, `x.toString()` ends with `".0"` for such values (e.g., `"5.0"`) whereas in Scala.js, they only display the integer value. This is necessary because `toString()` is defined on `Any`. Since, if statically typed as an `Any`, an `x` whose

---

[2]Although JavaScript VMs sometimes need to internally box primitives, they will do so consistently across Scala.js-emitted code and regular JavaScript code, which still allows for proper interoperability.

value is a `number` 5 cannot be dynamically determined to be an `Int` or a `Double`, `toString()` must return the same string in both cases. The natural choice is to return what JavaScript returns for such values, i.e., `"5"`, without the trailing `".0"`.

Again, we had to validate this change against the ecosystem. This turned out to be surprisingly more troublesome than the type-tests themselves, because of unit tests. A large number of tests in various libraries are testing the `toString()` of their containers with `Doubles` that happen to have whole values (for example, an `Option(5.0)`). The resulting strings differ on the JVM and on JS, hence the tests break, although the underlying logic was correct. We have been fortunate that library maintainers have been willing to adapt the tests to avoid such values.

Finally, besides primitive numeric types, Scala has one more primitive type whose semantics changed because it is not boxed: `Unit`. The `toString()` of the unit value `()` is `"()"` on the JVM, but has been respecified as `"undefined"` in Scala.js, so that `()` can interoperate with JavaScript's `undefined` value. This makes sense because `()` and `undefined` are the values that are returned by methods that do not explicitly return a value, respectively in Scala and in JavaScript. Should we have the luxury to redesign Scala for better portability across the JVM and JS, we could also specify that `().toString()` should return `"undefined"` on the JVM.

2.1.3 Run-time reflection

Although run-time reflection is provided as libraries in Scala (either Java's reflection API or Scala's own), it is often thought of as a language feature, since it is provided by the JVM. While it would certainly be possible to reimplement it entirely in JavaScript, this has one major issue: in the absence of VM support, reflection prohibits many ahead-of-time optimizations, in particular inter-procedural dead code elimination. In the JavaScript world, dead code elimination is an absolute must-have, because the size of the resulting bundles matters. Therefore, portability needs to yield to another property: the sheer ability to use the Scala.js language in contexts where JavaScript is used, without insane overheads of code size. For that reason, Scala.js does not support run-time reflection.

However, as with most things, there are part of run-time reflection that are critical to libraries in the Scala ecosystem. In particular, any dialect of Scala is effectively forced to support the following features:

- The method `java.lang.Object.getClass()`, returning the `java.lang.Class[_]` of the receiver
- The class `java.lang.Class[A]` itself and a few of its methods, namely
    - `isAssignableFrom(that: Class[_]): Boolean`
    - `isInstance(obj: Object): Boolean`
    - `isArray(): Boolean`
    - `getName(): String` and `getSimpleName(): String`

```
        – getComponentType(): Class[_]
        – cast(obj: Object): A
   • A few methods of java.lang.reflect.Array:
        – newInstance(componentType: Class[_], length: Int): Object
        – newInstance(componentType: Class[_], dimensions: Array[Int]): Object
        – getLength(array: Object): Int
        – get(array: Object, index: Int): Object
        – set(array: Object, index: Int, value: Object): Unit
```

This is the minimal amount of reflection utilities that are required by the Scala language and standard library. Most of it is mandated by `ClassTags` in Scala, and their API to build arrays of reified generic types.

Dropping run-time reflection is of course a serious threat to portability. It outright prevents some Scala libraries from being directly supported, and some use cases are even impossible to port not to use some amount of reflection. Besides the standard library itself, which requires the above methods, we found the following inherent use cases for run-time reflection in the ecosystem:

  • the ability to find a concrete class by its name, and if it extends a given superclass/trait, invoke one of its constructors, and

  • the ability to find a top-level `object` by its name, and if it extends a given superclass/trait, load its singleton instance.

A typical example of these requirements is testing frameworks. They need to be able to instantiate a test class given its name, but they know that each such test class will inherit from a given trait, say `Test`. On the JVM, these requirements are implemented on top of `Class.forName`, `Class.getConstructors`, etc. It is problematic to offer these methods in Scala.js for the reason presented above, but a slightly different API that covers precisely those use cases can be provided, in the package `scala.scalajs.reflect`. This API requires that a superclass (`Test` in this case) be annotated with `@EnableReflectiveInstantiation` for a class or object to be reflectively discoverable, which means it is not as powerful as the general `Class.forName`.

This API, being specific to Scala.js, cannot directly be used to write portable code. However, we can build a portable library on top of `reflect` on Scala.js and on top of `Class` methods on the JVM, with a common public API. This has been done in https://github.com/portable-scala/reflect.

Combining those various APIs, as well as the structural types presented in the following section, covers all the use cases of run-time reflection that we have found in the Scala ecosystem. Of course, fully supporting run-time reflection would enable even more libraries to be portable, so there would still be value in it, but it does not appear that the added value would compensate

the problems in terms of optimizations, and in particular dead code elimination.

2.1.4 Structural types

The Scala type system features structural types, as in the following example:

```scala
def callFoo(o: Any { def foo(x: Int): Seq[Int] }): Seq[Int] = o.foo(42)

class Bar {
  def foo(x: Int): List[Int] = List(x, x * 2)
}

callFoo(new Bar)
```

Even though there is no named interface for the parameter of `callFoo`, we can pass it a `Bar` because it *structurally* conforms to the parameter type. Note that the result type of `Bar.foo` is a subtype of the one declared in the structural type, but the parameter list must be equivalent (as applies to all method overrides in Scala and Java). This feature is well understood in terms of type theory, but is somewhat problematic to implement on the JVM, because it only supports nominal virtual dispatch. To circumvent that limitation, the code generated by the Scala compiler for the JVM uses a non-trivial amount of run-time reflection to perform the call. However, as we saw in the previous section, Scala.js does not support general reflection.

It turns out that supporting this feature of Scala is particularly difficult in the well-typed, whole-program view that Scala.js uses in its IR. We will see in Section 3.2.6 that we have to provision for that specific feature right inside the IR. This is a non-negligible burden on the definition of the IR and the linker implementation, but it has one advantage: it provides some amount of controlled run-time reflection that the linker knows how to handle. Anecdotally, this has proved to be useful as a substitute for run-time reflection in specific cases.

2.1.5 Compile-time overloading

Perhaps the single most impactful portability concern is that of overloading. In Scala, overloading is resolved at compile-time based on the static types of the actual arguments. By contrast, in JavaScript we encode "overloading" as run-time dispatch code, based on the *dynamic* types of the actual arguments. Trying to naively encode one into the other can cause a different behavior at run-time, hence is a threat to portability. For example, consider the following Scala code, simplified from a real-world example in the library:

```scala
class StringBuilder {
  var content: String = ""

  def append(obj: Object): Unit =
    content += obj
```

```
  def append(s: CharSequence): Unit =
    append(s: Object)
}
new StringBuilder().append("hello")
```

The `append` method is overloaded based on the type of its argument. With compile-time overloading the call to `append(s: Object)` systematically refers to the first definition of `append`, which is the correct behavior. If however we try to encode the above code with run-time dispatch in JavaScript as follows:

```
class StringBuilder {
  constructor() {
    this.content = "";
  }
  append(x) {
    if (x instanceof CharSequence)
      this.append(x);
    else
      this.content += x;
  }
}
```

we can immediately see that it will cause an infinite recursion when called with an instance of `CharSequence`.

Of course, encoding compile-time overloading is a solved problem. We can easily fix it with name mangling, giving different names to the overloads. An obvious encoding is to use the full (erased) signature (parameter types and result type) into the JavaScript name, since this is how overloading is encoded on the JVM, thereby maximizing portability:

```
class StringBuilder {
  constructor() {
    this.content = "";
  }
  append__Object__Unit(obj) {
    this.content += x;
  }
  append__CharSequence__Unit(x) {
    this.append__Object__Unit(x);
  }
}
```

which solves our issue.

This is in fact how Scala.js encodes compile-time overloading. However, the encoding poses a threat to interoperability instead, as an external JavaScript snippet that whishes to call the `append` method has to know about the specific encoding used by the compiler! Worse, if there is pre-existing JavaScript code that needs to be provided with an object that has an `append` method (not `append__Something`), we are incapable of implementing that interface from Scala.js code. This turns the threat to interoperability into a complete failure: if a JavaScript

library expects us to do that, we *cannot use* that library from Scala.js at all.

We will see in Section 2.2.5 how we design the interoperability features to solve this conundrum.

2.1.6 Compromising in the Name of Performance: Undefined Behaviors

We have argued that portability is very important to build a healthy ecosystem. However, sometimes we must sacrifice some amount of portability in the name of performance. As running example, consider dynamic downcasts, i.e., `x.asInstanceOf[C]`. In Scala/JVM, such a cast deterministically throws a `ClassCastException` if `x` is non-null but not an instance of `C`. It is definitely possible to implement the same behavior when targeting JavaScript. We simply define, for each class or interface `C`, a function of the form:

```
function asInstanceOf_C(x) {
  if (x !== null && !isInstanceOf_C(x))
    throw new ClassCastException();
  return x;
}
```

which we call for each occurrence of `x.asInstanceOf[C]` in the program. This is in fact what was implemented in Scala.js up to the 0.5.x series, included.

However, as we will see in Section 4.5, in some benchmarks, those checks are responsible for 50% or more of the total execution time, which means 100% overhead on average! Although Scala encourages a style where explicit casts are rarely necessary (due to its advanced type system and pattern matching), the semantics of erasure force the compiler to generate a lot of synthetic casts. For example, accessing an element of a `List` requires a cast:

```
def firstString(l: List[String]): String = l.head
```

is rewritten by the compiler as

```
def firstString(l: List): String = l.head.asInstanceOf[String]
```

On the JVM, these casts receive excellent support from the VM to be dynamically optimized away as much as possible. But on a JavaScript VM, the checks are encoded as user-space conditions, which the VM does not understand as well, and therefore cannot optimize.

Removing the checks seems like it solves a dramatic performance problem, but … how is that any different than the 64-bit integer issue? Why did we go to so much trouble to implement correct `Long`s (with a 3–5x slowdown) whereas we are easily dismissing correctness of casts because of a mere 2x slowdown?

First, the 3–5x slowdown of `Long`s only applies to subsets of the codebase using `Long`s. On the other hand, the 2x slowdown of `asInstanceOf` is ubiquitous. It contaminates the entire codebase.

Second, we consider that relying on the 64 bits of precision of a `Long`, and even on its arithmetic modulo 2^{64}, is admissible in a *correct* program. By contrast, encountering a failing downcast, which should throw a `ClassCastException`, is considered a programming error. Why should we penalize all correct programs with a 2x slowdown, in order to preserve the behavior of *incorrect* programs?

Of course, deciding what is or is not a programming error is more or less a philosophical question. When designing a cross-platform language from scratch, the language designer has the luxury to choose and impose their views. In the context of Scala.js, for which the JVM ecosystem already existed, we had to take into account existing codebases. It turns out that the existing Scala ecosystem virtually does not rely on `ClassCastException` being thrown (and caught) in correct programs, which allowed us to consider it as a programming error.

Undefined Behavior

If we consider `x.asInstanceOf[C]` to be a programming error when `x` is not `null` nor an instance of `C`, how we do specify it at the semantics level so that the compiler is allowed to remove the checks? A natural way would be to specify `x.asInstanceOf[C]` as always succeeding. This would satisfy both of the following properties:

- It preserves the behavior of correct programs, i.e., those where `x` is always either `null` or an instance of `C`, and

- It can be compiled to efficient code—an understatement, since it compiles to a no-op.

However, under that specification, the result of `x.asInstanceOf[C]` is not guaranteed to be a valid value of the type `C`. This breaks soundness of the type system, even with the erased semantics. This is problematic when it comes to specifying the effects of field access and method dispatch. As a consequence, an optimizer cannot rely on the static type of any variable, which prevents even the most basic optimizations such as inlining of monomorphic methods.

The alternative, widely used in low-level programming languages such as C and C++, is to specify incorrect casts as *undefined behavior*. Under that model, the program can behave arbitrarily when `x.asInstanceOf[C]` should fail.

Undefined behaviors are however notoriously difficult to grasp and program with. Moreover, they are virtually nonexistent in the life of a Scala/JVM developer, who is therefore not used to correctly deal with undefined behavior. Hence, introducing undefined behavior in the language is very dangerous, and the danger needs to be mitigated. To that effect, Scala.js features a *development mode* and a *production mode*.

In production mode, the compiler removes the checks for maximal efficiency. In a typical edit/compile/test cycle, however, a developer uses the development mode of the language and compiler. Under that mode, all would-be undefined behaviors are checked, and reliably

throw. They do not throw the expected type of exception, though (e.g., `ClassCastException`), as that would allow tests for code that catches those exceptions to succeed, whereas they would fail in production mode. Instead, the expected exception is wrapped inside a specific throwable class, named `UndefinedBehaviorError`, which must not be caught. That exception extends `java.lang.VirtualMachineError`, which is the general supertype of exceptions that cannot be reliably caught. This also means that `UndefinedBehaviorError` exceptions are not matched by the special `scala.util.control.NonFatal` extractor.

Some recent C compilers feature similar switches. For example, Clang has its UndefinedBehaviorSanitizer, with a wide range of switches to check various sources of undefined behavior [64]. In Scala.js, however, leveraging the checks for undefined behavior is part of the default workflow, encouraged both by the toolchain itself and by the documentation. As a result, every Scala.js developer uses the checks on a regular basis, providing much greater confidence that their codebases comply with the specification than is found in typical C codebases.

Despite those precautions, removing checks in production mode is dangerous. As Tony Hoare described it in his presentation "Null References: The Billion Dollar Mistake", a production mode is "like wearing a life jacket on your practice emergency drills, and taking it off as soon as your ship was really sinking."[3] Fortunately, in the case of Scala.js, even the effects of triggering an undefined behavior are bounded by the capabilities of JavaScript: the program cannot escape the security protections established by the JavaScript VM. This considerably limits the impact of undefined behavior on security concerns, compared to lower-level languages like C. Finally, for sensitive applications where that is not enough, the developer still has the option to leave the checks on using the appropriate configuration.

Alternatives

Instead of declaring programming errors as undefined behaviors, we could explore alternatives trying to reduce their overhead. Here are some avenues that we have considered but rejected in the context of Scala.js.

First, we could consider some kind of profiling to detect which casts always succeed, and which ones occasionally fail, on the grounds that we could remove the checks for the successful ones. While it would probably solve the performance issue, such a strategy does not improve safety compared to the hard undefined behaviors. Indeed, profiling must be done ahead of time, using known or random inputs. If it finds any failing cast, then a test suite using the same inputs would also have discovered the bug, which could have been fixed. The programming errors not identified by the test suite are still considered "safe" and hence unchecked, which still leaks the issues.

Another possibility would be to improve static analyses in order to prove that some casts and

---

[3]https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

other programming errors cannot happen. Program locations that are proven safe would not need the run-time checks. This alternative is safe, but it is questionable whether it would solve the performance issue. In practice, most casts in a Scala program are introduced by erasure, especially when taking values out of generic data structures. In these cases, proving that the casts are safe requires a very sophisticated points-to analysis that includes the heap, and is capable of "specialising" the data structures on the heap so that not all `Lists` in the program are analyzed as containing anything. We have not explored the feasibility of such an analysis.

Finally, we could leave checks corresponding to casts present in the source code, while removing those introduced by the compiler due to erasure, on the grounds that the latter are always safe (assuming the Scala type system is sound). This, unfortunately, does not hold, as illustrated by the following snippet:

```
val intList: List[Int] = List(1)
val stringList: List[String] = intList.asInstanceOf[List[String]]
val string: String = stringList.head
```

which, after erasure, becomes

```
val intList: List = List(1)
val stringList: List = intList.asInstanceOf[List]
val string: String = stringList.head.asInstanceOf[String]
```

In this example, the source-level cast on the second line succeeds, while the compiler-emitted cast on the third line fails. Only checking source-level casts therefore does not fundamentally improves safety compared to removing all checks.

2.1.7 Evaluating Portability

In order to evaluate the extent to which Scala.js is portable with respect to Scala/JVM, there is but one source of truth: tests. As part of the test suite of Scala.js, we run the tests of Scala/JVM itself, both the so-called partest and the JUnit tests. Each test can fall into one out of four levels of compatibility:

- Correct: the test successfully passes.
- Amended: the test succeeds if its expected output is amended to match some altered semantics (such as `toString()` of primitive values).
- Ignored: the test is blacklisted because it tests an area of the language that is not supported at all (for example, run-time reflection).
- Incorrect: the test fails, although it should pass (these are known issues).

Table 2.1 shows the amount and percentage of tests in each category, for partest and JUnit, with respect to Scala 2.12.5. The most common reasons to ignore tests are using run-time reflection, testing the compiler API, and using mixed compilation of .java and .scala source files.

	Correct		Amended		Ignored		Incorrect	
partest	3365	(78%)	54	(1%)	905	(21%)	1	(0.02%)
JUnit	40	(25%)	0		121	(75%)	0	

Table 2.1 – Amount and percentage of tests from Scala/JVM falling into various compatibility buckets

	Portable		JVM-only	
Shapeless	598	(96%)	23	(4%)
scalaz	7677	(98%)	218	(2%)
cats (core)	7541	(99%)	46	(1%)

Table 2.2 – Portable tests versus JVM-only tests in major cross-compiling libraries

In addition to the tests for the compiler and standard library, we also look at some major cross-compiling libraries and at how many of their tests are cross-compiled (hence, portable) compared to those restricted to the JVM. Table 2.2 shows the result. We can see that popular libraries expose a higher percentage of portable tests than the compiler and standard library. This is easily explained by the large number of tests for run-time reflection in the standard library, whereas the Scala ecosystem tends not to use run-time reflection in practice, preferring compile-time reflection (macros) instead.

These three libraries were initially developed without support for Scala.js. Shapeless and scalaz, in particular, had been around long before Scala.js was created. They added support of Scala.js after the fact, once it became portable enough. From the commits that introduced support for Scala.js in scalaz[4] and cats,[5] we can see that changes are mostly limited to the build infrastructure and not to actual source files. Some files are moved to the JVM-only directories, typically those involving blocking operations (`FutureInstances.scala` in scalaz or `FutureTests.scala` in cats), since they are fundamentally incompatible with JavaScript's memory model.

2.2 Expanding the Universe: Interoperability

In the previous section, we have explored how language semantics support portability. This is enough to cross-compile Scala code that does not interact with its environment. However, to be able to use JavaScript libraries, including interacting with the host environment's bindings such as the DOM or the Node.js APIs, we need interoperability.

The work presented in this section has been done in collaboration with Tobias Schlatter and Nicolas Stucki, and published in the proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala [17]. We have updated it to some of the new interoperability features that have

---

[4]https://github.com/scalaz/scalaz/commit/d63dcaeeb444ef4e2609df5b181326a12d5d5fcb
[5]https://github.com/typelevel/cats/commit/009fee481f54e3e3529e548b73686b5eae319ea1

been added in the language since the publication of the paper, notably imports and exports, references to global variables and creation of class values with captures at run-time.

Looking at other languages compiling to JavaScript, we can separate them into two main categories: languages with the same run-time semantics as JavaScript, and languages with different run-time semantics (such as compile-time overloading).

In the first category, we find languages such as CoffeeScript [10], TypeScript [45] and Flow [21]. In those languages, talking to JavaScript APIs is easy, since there is virtually no impedance mismatch between the languages. At most, it requires defining type definitions for JavaScript APIs to be able to call them easily, as is the case in TypeScript.

In the other category, we find all the languages that either offer more powerful language abstractions (e.g., PureScript [53], Elm [19]) or support multiple target runtimes (e.g., Clojure-Script [9], GWT [25]). These languages all fail, at some level or another, to provide satisfactory interoperability with JavaScript APIs, as we will see in Section 2.2.1. Shortcomings range from the inability to handle object-orientation, overloading, or higher-order functions, to requiring knowledge of the implementation details of the compiler. If a language misses some interoperability features, there are some JavaScript libraries that it *cannot interact with*. Usually, this is worked around by writing "bridge" JavaScript code working as an adapter for the library, reexposing its functionality with a subset of the JavaScript language features understood by the program.

As we mentioned in Chapter 1, when designing Scala.js, interoperability with JavaScript has been the number one priority. An early design [14] was no better than the state of the art (it suffered from issues similar to those we discuss in Section 2.2.1), and prompted us to research a different approach to interoperability. While part of the second category of languages, Scala.js now solves the shortcomings of previous approaches to interoperability. It does so with what we call a semantics-driven approach: provide Scala.js language features for all the run-time semantics offered by the ECMAScript 2015 language, so that Scala.js programs can literally do everything that ECMAScript programs can do. This guarantees that Scala.js programs can talk to *any* JavaScript library. At this stage, the semantics for one ECMAScript construct, namely `new.target`, is still being worked on, which means this guarantee is not actually provided by the currently implemented system.

2.2.1 Motivation

We first present some shortcomings of previous approaches to interoperability with JavaScript. We give most examples in GWT and ClojureScript, because they are, in our opinion, among the languages providing the best interoperability features. Their few shortcomings are the most challenging ones to tackle. They are not fundamental limitations, but rather consequences of the current design of interoperability in these languages. Future versions of ClojureScript or GWT could address those concerns, possibly with the approach described here.

Overloading   Although JavaScript technically does not have overloading as a language feature per se, the term "overloading" is commonly used in JavaScript—including in its specification, e.g., [18, §20.3.2]—as referring to run-time dispatch based on run-time tests, both for the actual number of arguments provided at call site (arity-based overloading) and for the types of the actual arguments (type-based overloading).

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

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

In a language whose interoperability features cannot express overloading, such as GWT in their latest version [26], calling such an API is not directly possible. Workarounds include a) writing a bridge JavaScript library that exposes `getHTML()` and `setHTML()` separately, or b) using two different types for `list` (one declaring the getter, the other the setter, with an explicit cast in between). Neither workaround is satisfactory. Besides, they do not allow a GWT program to *implement* such an API, to be consumed by another JavaScript module. It is impossible to write a class exposing an overloaded method to JavaScript, as mentioned in Section "Caveats & Special cases" of the GWT interoperability specification [26].

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

Object-Orientation   Consider the following JavaScript code using Phaser [39], a game development library. It creates a very simple game state that draws a triangle on the screen.

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

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

old-fashioned way:

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

However, this workaround would stop working if Phaser migrates `Phaser.State` to an actual ECMAScript 2015 class. Indeed, it is not possible to extend an ES 2015 class from an ES 5.1 constructor function.[6]

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

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

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

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

Conclusion As we have shown in this section, existing interoperability solutions have severe limitations. When some constructs available to JavaScript applications are impossible to reproduce in a source language, there are JavaScript libraries that *cannot be used* by applications written in that language. It is therefore important to design interoperability features that avoid this situation. This can only be achieved if those features allow to reproduce any behavior that could be achieved in JavaScript, i.e., if they cover "all of JavaScript". We will make this criterium more precise in Section 2.2.6, but will first go through the design of interoperability

---

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

in Scala.js.

2.2.2 Scala Types and JavaScript Types

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

How can we address the semantics mismatch in Scala.js? Brutally changing that aspect of the Scala semantics when compiling to JavaScript is not acceptable, since it would break valid programs, as we saw in Section 2.1.5. Our solution to this problem is the following: do not shy away from the semantics mismatch, but rather acknowledge its existence, and encode it into the type system. We do this with a separate hierarchy of *JavaScript types*, rooted in the type `js.Any`, a third subtype of `Any` (beside `AnyVal` and `AnyRef`, which are Scala types). Unlike Scala types, JavaScript types have *JavaScript semantics*.

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

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

Because `list` has a JavaScript type, calling `list.html()` is intuitively equivalent to the corresponding JavaScript code, i.e., it looks for a property named `html` in the prototype chain of `list`, checks that it is callable, and calls it with `list` as value for `this` and zero argument. Similarly, `list.html(<expr>)` calls it with one argument, which is the result of evaluating `<expr>`. We will make the semantics of method calls more precise in Section 2.2.4.

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

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

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

Table 2.3 – Type correspondence in Scala.js

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

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

2.2.3 Type Correspondence

The attentive reader might have noticed that we glossed over an important detail in the example from Section 2.2.2. We have happily assumed that `String` accurately represents the values that jQuery expects and returns. If it doesn't, our previous code would be flawed, as it would call jQuery's `html()` method with something that it cannot understand.

Recall from Section 2.2.1 that GWT's interoperability specifies that a primitive `double` is seen by JavaScript as a primitive `number`, but that boxes thereof are not. In Scala.js, the language mandates the type correspondences shown in Table 2.3, regardless of whether those values enter generic contexts (or are upcast to `Any`). In other words, Scala.js does not exhibit the boxing issue. Implementation-wise, values of those types are never boxed in Scala.js. The reader may recall from Section 2.1.2 that we had to sacrifice some amount of portability to gain this property. Instances of all other Scala types (including `Char` and `Long`) are specified as *opaque*. JavaScript sees them as objects with which it cannot interact. The system implemented in Scala.js allows to partially open up Scala types to JavaScript with *member exports*, but a discussion of member exports is omitted from this dissertation.

2.2.4 Manipulating Values of JavaScript Types

Method Calls

Recall from Section 2.2.2 that method calls on JavaScript types have run-time overloading dispatch. More importantly, they have JavaScript method call semantics. Intuitively, this means that a Scala.js method application of the form `x.meth(a, b)` where `x` is statically typed as a JavaScript type behaves as the JavaScript code `x.meth(a, b)`.

Past the intuition, however, this does not make sense, as `x`, `a` and `b` can be arbitrary Scala.js expressions, which are not valid JavaScript expressions in general. A better definition of the semantics of such a call would be: evaluate the Runtime Semantics of `x.meth(a, b)` according to the ECMAScript specification [18, §12.3.4], replacing invocations of `GetValue(x)` (resp. `a`, `b`) by the evaluation rules of the Scala language specification [48, §6] for x (resp. a, b). However, a completely formal derivation of the evaluation rules is out of the scope of this dissertation. Chapter 3 provides a formal specification for the intermediate representation of Scala.js, although not for Scala.js source code itself. We will therefore stick to the less formal definition of the semantics for the remainder of this chapter, implying that evaluation of subexpressions is left to the Scala semantics, unless otherwise noted.

Result Values: Protecting our Borders   Recall from Section 2.2.2 that the `html()` method is declared with an explicit result type of `String`. The Scala type system therefore expects and believes that calling `html()` will return a `String`. With JavaScript semantics, however, this might not be the case. The method could, for example, return an `Int`, if it is implemented in a native JavaScript class. If this happens, statically typed Scala code will start manipulating an `Int` value as if it were a `String`. This could have disastrous consequences. It is also extremely damaging for an optimizer, which cannot rely on a sound static type system to perform optimizations, even on Scala types.

To avoid these problems, we protect our borders by systematically checking that the values returned by JavaScript method calls conform to the static result type, in the fashion of [65, 42] (though without blame tracking). In other words, if `html()` returns an `Int`, a `ClassCastException` will be thrown. Explained as a code example, the call `x.html()` is actually equivalent to `x.html().asInstanceOf[String]`.

Because of type erasure in Scala, run-time type checks are always performed up to the erasure of a type, as defined by the Scala language specification [48, §3.7]. This means that a `list` of type `List[Int]` qualifies as `List[String]` for the purpose of this check, which therefore succeeds. It is only later, when accessing an element of that list, e.g., `list.head`, that an additional check will be performed, as `list.head.asInstanceOf[String]`. In Scala and in Scala.js, types are therefore only sound up to erasure. Even though erasure is often regarded as a liability, it gives Scala.js immunity against the common performance problems found in

interoperability layers that check types at the borders [61]. Indeed, only the first-order type needs to be checked, which is a constant-time operation. Moreover, since typed function values already expose an untyped entry point due to erasure, Scala.js does not need wrappers for function values sent to dynamically typed parts, i.e., to JavaScript libraries, which means that function identity is not threatened.

In Scala.js, the expression `x.asInstanceOf[T]` is further specified as a no-op if the erasure of `T` is a JavaScript type. This also applies to manual `asInstanceOf` checks, as well as those automatically inserted for the erasure of generics. Consequently, *any* value can be successfully cast to any JavaScript type, making JavaScript types decidedly unsound (similarly to generic types). Operations applied to JavaScript types are always checked at run-time (by the JavaScript VM). This property makes JavaScript types behave similarly to the *like types* proposed by Wrigstad et al. [69], with the exception that casts from non-conforming types must be explicit.

Table 2.4 recaps all the interoperability semantics of expressions manipulating values of JavaScript types.

Function Call

Some JavaScript values are *callable*, i.e., they can be called with the function call notation `f(a1, ..., an)`. In Scala, a corresponding syntax exists, which desugars into calling the `apply` method of the function value, i.e., `f(a1, ..., an)` desugars into `f.apply(a1, ..., an)`. It is therefore natural to specialize the semantics of calling a method named `apply` into the semantics of a JavaScript function call. This allows to declare interfaces for JavaScript function values, similar to the ones for Scala function values. For example, a 1-argument function value can be typed as follows (including the variance annotations `-T1` and `+R`):

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

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

References to global variables and imports

So far, we have seen how to manipulate JavaScript values that we already have a reference to, but we do not have any means to import JavaScript values from the top-level entry points of other libraries. In JavaScript, there are two main sources of such entry points: global variables and imports.

Global variables are bindings in the global lexical scope. Until ECMAScript 5.1, all such bindings were also accessible as properties of the global object, but ECMAScript 2015 changed

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

Table 2.4 – Semantics of manipulating JavaScript types. `x` is assumed to have a static JavaScript type. For a result type R, results are cast to R as per `result.asInstanceOf[R]`. Each line of the table reads as: given a static declaration (in a JavaScript type) from the first column, the Scala.js expression from the second column has the same run-time semantics as the JavaScript code in the third column (where [e] denotes the evaluation of e according to the Scala specification). If x is an `@JSGlobalScope` object, `[x].y` is replaced by `global:y`. If a member y is annotated with `@JSName(n)`, `[x].y` is replaced by `[x][[n]]`.

the rules: top-level `lets`, `consts` and `classes` are only accessible in the lexical scope. It is therefore crucial for Scala.js to be able to access global bindings. We can annotate a top-level JavaScript `object` with `@JSGlobalScope` (and `@js.native`) to specify that its members should be looked up in the JavaScript global scope:

```scala
@js.native
@JSGlobalScope
object GlobalVariables extends js.Any {
  val document: HTMLDocument = js.native
  def parseInt(s: String): Int = js.native
}

val input = GlobalVariables.document.getElementById("age")
val age = GlobalVariables.parseInt(input.value)
```

Member selections such as `GlobalVariables.document` have the semantics of looking up the JavaScript identifier `document` in the JavaScript global scope.[7] Accessing a member whose name is not a valid JavaScript identifier is a compile error.

For global variables that are classes or objects, we can use the shortcut `@JSGlobal` instead, so that they can be written at the top-level of a Scala package, which gives a better Scala.js API:

```scala
@js.native
@JSGlobal
class Date extends js.Object {
  def getTime(): Double = js.native
}

@js.native
@JSGlobal
object Math extends js.Object {
  def sqrt(x: Double): Double = js.native
}
```

Such definitions are equivalent to wrapping them in an `@JSGlobalScope` object.

For imports in an ECMAScript 2015 module, we have a similar annotation `@JSImport`:

```scala
@js.native
@JSImport("fs", JSImport.Namespace)
object NodeFS extends js.Object {
  def readFileSync(path: String): Buffer = js.native
}
```

The first argument is the module name, and the second one is either

- `JSImport.Namespace`, mapping to `import * as Foo from "moduleName"`,

- `JSImport.Default`, mapping to `import Foo from "moduleName"`, or

- a constant string, e.g., `"foo"`, mapping to `import {foo as Foo} from "moduleName"`.

---

[7]Or, to be precise, in the lexical scope surrounding the code generated by Scala.js, which can be slightly different in some environments such as CommonJS modules in Node.js.

Top-level exports

Speaking of module imports, we can also export top-level objects, classes, variables and methods with the `@JSExportTopLevel` annotation. Since Scala does not have top-level variables and methods per se, those declared in a Scala top-level objects are considered as top-level for this purpose. Getters and setters cannot be exported to the top-level, since ECMAScript does not allow to export properties from a module.

```scala
@JSExportTopLevel("O")
object O extends js.Object

@JSExportTopLevel("C")
class C extends js.Object

object Container {
  @JSExportTopLevel("five")
  val five: Int = 5
  @JSExportTopLevel("config")
  var config: Any = null
  @JSExportTopLevel("square")
  def square(x: Int): Int = x * x
}
```

When emitting the Scala.js code as an ECMAScript Script, those correspond to top-level `const` declarations (or `let` for `var`s). When emitting as a Module, they correspond to `export` statements.

Other Features

We omit a detailed discussion of several additional interoperability features:

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

Symbolic names: JavaScript properties and methods can have names that are `symbols` rather than `strings`. They can be declared in JavaScript types as members with the annotation `@JSName(SymbolValue)` where `SymbolValue` is a global `val` holding the value of the symbol, e.g., `js.Symbol.iterator`.

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

Operators: Similarly to how methods named `apply` were mapped to JavaScript function calls in Section 2.2.4, methods whose name is one of the JavaScript symbolic operators are

mapped to the semantics of the corresponding operator. Alphabetical operators such as `typeof` are provided by primitive functions, e.g., `js.typeOf(x)`.

js.constructorOf: In JavaScript, classes are first-class terms, which can be manipulated as such. In Scala, however, they only live in the namespace of types. The primitive `js.constructorOf[x.C]` reifies the JavaScript class value corresponding to a class at the term level. It is notably useful when a library expects a constructor (the class value) as argument. `js.constructorOf` is also used to specify the behavior of `new` and `isInstanceOf` with JavaScript classes.

Together, and in addition to the features we have seen before, these features can be used to accurately type the API of JavaScript arrays, for example:

```
@js.native
@JSGlobal
class Array[A] extends js.Object {
  def length: Int = js.native
  def length_=(v: Int): Unit = js.native

  @JSBracketAccess
  def apply(idx: Int): A = js.native
  @JSBracketAccess
  def update(idx: Int, v: A): Unit = js.native

  @JSName(js.Symbol.iterator)
  def iterator(): Iterator[A] = js.native
}
```

2.2.5 Creating JavaScript Values

Whereas Section 2.2.4 exhaustively showed how we can manipulate values of JavaScript types, this section highlights the features of Scala.js' interoperability that allow to create JavaScript values.

Values of Primitive Types

There are six primitive types in JavaScript: `undefined`, `null`, `boolean`, `number`, `string` and `symbol`. With the exception of `symbol`, all of them have equivalent Scala types as defined in Section 2.2.3. The regular Scala constructs to create values of those types (such as literals) can be used to create the JavaScript values, as they are the same. Symbols are created as in JavaScript, using the functions `Symbol(desc)` and `Symbol.for(key)`, which can be called through the interoperability semantics for function call and method call.

Function Values

Even though Scala has syntax to create function values, they are Scala function values, which are not equivalent to JavaScript function values. We can however create a JavaScript function values off an anonymous function if the expected type is one of the `js.FunctionN`, since those types are Single Abstract Method (SAM) types. For example, we can create a JavaScript of one argument as follows:

```
val f: js.Function1[Int, Int] = a => a + 1
```

Formally, when an anonymous function `(p1, ..., pN) => e` is typed as a `js.FunctionN`, it evaluates to a new function object, as defined in [18, §9.3], that, when called, assigns the *N* first actual arguments to `p1…pN`, and returns the result of evaluating `e`. Such function values always discard the value of `thisArgument` that they are given, as do arrow functions in JavaScript. An additional series of `js.ThisFunctionN` types explicitly capture the `thisArgument` as an additional formal parameter at the Scala.js level.

For convenience, the standard library provides implicit conversions between Scala functions and JavaScript functions. For example, for functions of one argument, we have:

```
implicit def fromFunction1[T1, R](f: T1 => R): js.Function1[T1, R] =
  (x1) => f(x1) // creates a JS function because of the expected type

implicit def toFunction1[T1, R](f: js.Function1[T1, R]): T1 => R =
  (x1) => f(x1) // creates a Scala function
```

Class Values

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

Since most parts of class definitions straightforwardly map to corresponding concepts in JavaScript classes, we omit a detailed discussion, and summarize the semantic equivalences in Table 2.5. Note that formal parameters are checked with casts to protect the borders, since JavaScript code calling the method could give arbitrary parameters. If a JavaScript class is declared in a `def`, each invocation of the method creates a distinct JavaScript class value—unlike for a Scala class, which the compiler rewrites as a unique top-level class storing its captures as additional fields. Similarly, a JavaScript class declared in the body of a `class` creates a class value for each instance of the enclosing class.

One aspect deserves to be further discussed, though, namely overloading. In Section 2.2.2, we saw that overloading in JavaScript types has run-time dispatch semantics. Section 2.2.4

Scala.js definition	Semantics as JavaScript code
```class C extends D with ...```	```class C extends ⟦js.constructorOf[D]⟧```
```val f:T``` ```var f:T```	```this.f = ⟦defaultOf[T]⟧```, in the constructor

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

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

In the companion object:
```@JSExportStatic```
```def m(p1:T1,..., pn:TN):R = ...```
(similar for fields, getters and setters)
	```static m(q1,..., qn) { ... }```

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

Table 2.5 – Semantics of the definition of JavaScript classes (without overloading considerations).

provided precise semantics for method calls, but we now need to give semantics to method definitions. Consider the following definitions:

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

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

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

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

1. Switch on the number of actual arguments for each group of overloaded definitions with the same number of formal parameters.

2. For each value of the number of formal parameter count:

   (a) If there is only one overloaded definition, call it, with appropriate type checks for the actual arguments.

   (b) Otherwise, find the first parameter position at which the type is not the same in all definitions. Perform run-time type tests on that parameter to refine the set of possible definitions, and go back to step 2a.

   (c) If all the erased types are equal, throw an error. The existence of this case can be decided at compile-time, and is reported as a compile error.

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

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

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

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

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

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

2.2.6 Completeness

Now that we have gone through all the interoperability features of Scala.js, it is time to revisit our initial goal. Recall from Section 2.2.1 that, in order to be able to talk to any JavaScript libraries, we need our interoperability features to cover "all of JavaScript". We argue that our interoperability features are *complete*, in the sense that they support "all of JavaScript" (in Strict Mode [18, §10.2.1]). But what exactly *is* "all of JavaScript"? We find the answer in the ECMAScript specification [18], Sections 10 to 15, included, which are entitled "ECMAScript Language". Supporting "all of JavaScript" essentially means being able to express all of the run-time semantics offered by the ECMAScript language.

Table 2.6 shows a comprehensive list of the relevant subsections in the ECMAScript specification, together with links to the sections describing the corresponding interoperability features. The ECMAScript specification defines both static and run-time semantics. The former being more about JavaScript source code than evaluation semantics, only the latter are relevant. In particular, we entirely skip Sections 10 and 11 entitled "Source Code" and "Lexical Syntax", respectively. We also skip language constructs which are obviously replaced by corresponding Scala language constructs, such as identifiers and control structures.

As we can see, there are exactly two features that are not implemented: `new.target` and generator functions. Generator functions are syntax sugar over normal functions. If needed,

ECMAScript 2015 language features		Scala.js interoperability features
12.2.2	this	this in methods; js.ThisFunction in functions
12.2.4	literals	Scala literals and type correspondence
12.2.5	array initializer	js.Array(e1, ..., en), implemented with function call
12.2.6	object initializer	new js.Object { val f1 = v1; ... }, an anonymous class
12.2.8	regex literals	can be implemented with lazy val re = new js.RegExp("re", "flags")
12.3.2	property accessors	property access and indexed properties
12.3.3	the new operator	new for JavaScript classes
12.3.4	function calls	method calls and function calls
12.3.5	the super keyword	super references
12.3.8.1	new.target	*not supported*
12.4.{4–5}, 12.5.{7–8}	++ and ––	can be implemented with += 1 and –= 1
12.{4–12}	other operators	JavaScript operators and primitive methods
13.7.5	for–in loop	primitive js.special.forin()
13.7.5	for–of loop	implemented as user code in the Scala.js standard library
13.16	the debugger statement	primitive js.special.debugger()
14.1	function definition	anonymous functions with expected type js.FunctionN
14.2	arrow function definition	anonymous functions with expected type js.ThisFunctionN
14.3	method definitions	methods, getters and setters in JavaScript classes
14.3	static methods and properties	@JSExportStatic members in the companion object of JavaScript classes
14.4	generator function definitions	*not directly supported*, but they desugar into other concepts that are supported
14.5	class definitions	class value definitions
15.2	modules	@JSImport and @JSExportTopLevel

Table 2.6 – ECMAScript 2015 language features and the corresponding interoperability features of Scala.js. Each line of the table reads as: the run-time semantics of the JavaScript construct from the first column (as specified in the referenced section of [18]) can be expressed in Scala.js using the interoperability feature from the second column.

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

`new.target`, on the other hand, is a truly missing piece of semantics that cannot be otherwise represented. Although it would be technically very easy to implement at this point, we are still debating on the best way to expose it in Scala.js source syntax.

2.2.7 Related Work

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

Matthews and Findler [42] develop a theoretical foundation for interoperability between a statically typed language and a dynamically typed language. They define a Natural Embedding that allows to embed Scheme-like programs into ML-like programs and vice versa. At the boundaries, numbers and functions are converted on a type-directed basis between the Scheme representations and ML representations, with dynamic type tests to ensure that values match their types when flowing from Scheme to ML. Function values are only checked up to their first-order behavior, delaying further checks for the arguments (or result values) to the application of the functions. We do something very similar in Scala.js: values coming from JavaScript semantics and flowing into statically typed Scala code are downcast to their erasure, which is essentially equivalent to Matthews and Findler's first-order behavior. Dynamic checking of static contracts is also a recurring theme in gradual typing [58, 65], although the run-time semantic difference issue does not apply in those cases. Gradually typed systems often suffer from performance problems [61] due to either deep type tests or towers of wrappers. Scala.js avoids those issues by blending type tests related to interoperability borders with those necessary due to erased generic types in Scala.

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

Type-based representations of foreign language mutable objects has been the topic of several practical systems, however. Some early work was done by Elsman in SMLtoJs [20], using phantom types to represent JavaScript values. That system considered foreign values as

blackboxes that could not be directly manipulated by SML code. More recent developments include the overlay types and JsTypes of GWT [25], which do allow direct manipulation. However, as we saw in Section 2.2.1, their semantics have several limitations due to their implementation-based behavior.

Independently of the run-time semantics problem, typing native APIs with Scala.js traits, classes and objects was initially heavily inspired by TypeScript's type definitions [45]. We adapted some language features to fit Scala's type system, such as using `@JSBracketAccess` annotations where TypeScript has dedicated syntax. We later expanded the framework to support the definition of class values from Section 2.2.5.

2.2.8 Conclusion

We have shown how we can extend Scala's type system with an additional type hierarchy for JavaScript types. Operations on values of these types have different run-time semantics than traditional Scala values, namely JavaScript semantics. This change of semantics closes the impedance mismatch between Scala and JavaScript, allowing to talk to JavaScript APIs from Scala.js code. We have argued in Section 2.2.6 that our interoperability features are virtually complete with respect to the ECMAScript 2015 language: they offer all the run-time semantics that are offered by ECMAScript. One missing feature, namely the meta-property `new.target`, is yet to be covered, although its omission is due to being undecided on the best source syntax rather than to an actual technical challenge. We say that this approach is semantics-driven because it provides language constructs in Scala.js that have the semantics of ECMAScript constructs, so that "all of ECMAScript" is supported in Scala.js. To the best of our knowledge, Scala.js is the first system providing "all of ECMAScript", by that definition.

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

The interoperability semantics presented in this section have been used by the community to write type definitions for a large number of JavaScript libraries, including jQuery, React, Angular and the Node.js APIs,[8] which allows all these libraries to be called from Scala.js.

---

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

2.3   Lessons learned

In this chapter, we have explored the design of Scala.js, focused on portability wrt. Scala/JVM and interoperability with JavaScript. Besides the specifics of Scala and JavaScript, there are a few design strategies that can be reused for other cross-platform languages.

First, regarding portability, a language designer needs to decide which language features must be faithfully replicated, which ones can be amended with slight differences, and which ones are dropped or severely altered. The rule of thumb is that everything should be faithfully preserved, as that is the only way to guarantee that existing code behaves the same way across platforms. There are however exceptions:

- Interoperability concerns can be a compelling reason to slightly alter some of the semantics. For example, the instance tests on primitive numeric types, and their `toString()`, are different than Scala/JVM, so that numeric types can be compatible with JavaScript primitive numbers, improving interoperability.

- Some erroneous behaviors can be altered, either by throwing different kinds of exceptions or by being turned into undefined behavior. This is only possible for scenarios that are considered "programming errors", which is, unfortunately, ultimately a judgment call. A loose definition is that a programming error cannot happen in a bug-free program, which means that bug-free programs are portable.

- In some cases, entire language features can be dropped, if they are impractical to implement on a platform, and not widely relied on in a particular language ecosystem. The obvious example in the case of Scala.js is run-time reflection: some isolated functions are supported because they are widely relied on, but most capabilities of run-time reflection in Scala/JVM are outright dropped in Scala.js. For dropped language features, problematic programs should ideally not compile (or not link).

When designing a new language, the borders between those categories are fairly maleable. However, when porting an existing language, it is important to consider the existing ecosystem of libraries, in order not to prevent large subsets of the ecosystem to support the new platform. This can constrain some decisions, such as what erroneous behaviors can be considered programming errors in practice.

Regarding interoperability, we have argued in Section 2.2.6 for an objective measure: interoperability features must be complete with respect to the semantics of the host language. This ensures that we can talk to any library of the host language's ecosystem. Indeed, whatever the contract established by a library, it cannot demand of its users more than what the host language can express. Ensuring that our cross-platform language can express all the semantics of the host language therefore guarantees that it can abide by the contract of any library.

The designer of a cross-platform language must carefully balance all of those aspects to allow for the emergence of a corresponding cross-platform ecosystem.

Chapter 3

The Scala.js IR: A Simple Language with Portability and Interoperability

In the previous chapter, we have extensively studied the semantics of Scala.js as a source language. In particular, we have studied how it supports portability and interoperability. What we have not yet discussed is how we can reason about those semantics, and those two important properties, so that a compilation pipeline can understand the codebase well enough to compile and optimize all of it to JavaScript, while respecting the semantics.

In this chapter, we present the Scala.js Intermediate Representation, or SJSIR, which is a midway step from source language to optimized JavaScript code. In terms of level of abstraction, it is similar to .class files for the JVM:

- it is statically typed, with an erased type system,

- its type system is (expected to be) sound,

- its type system features classes, primitive types and arrays, and

- it is verifiable.

It also has significant differences, related to the specificities of a JavaScript target:

- the body of methods is represented as a typed expression tree, rather than using a stack-based representation,

- it has structured control flow instead of `goto`s, and

- its has dynamically typed JavaScript operations in addition to the statically typed JVM-style operations.

Just like a normal language, the Scala.js IR is defined by a syntax, a type system and run-time semantics. Together, they give a precise specification of the IR (more precise than the Scala.js source language specification), which acts as a contract between compiler front-ends emitting

the IR and back-ends compiling it down to JavaScript. While it is mostly used to compile Scala.js source code, it can be targeted by other language front-ends, such as Kotlin, for which we have a proof a concept.[1] The various versions of Scala can also be considered as separate front-ends: although all minor versions of Scala 2.x share their compiler to the Scala.js IR, the upcoming Scala 3.x will need a separate front-end. The contract induced by the specification is also the basis for optimizations: the optimizer transforms an IR program into another IR program that is semantically equivalent.

In addition to the above synchronous uses of the IR specification, there are diachronous advantages to having a well-specified IR. Indeed, the binary artifacts published on central repositories such as Maven Central contain the compiled IR, rather than the source code. As long as we evolve the specification of the IR in backward binary compatible ways, published libraries keep working as time progresses and new versions of Scala.js are released. This is important to build a large ecosystem of libraries, as library authors are not forced to recompile and republish their libraries with every new version of Scala.js. In order to fully leverage this advantage, the file format in which the IR is encoded must also allow future compatible evolutions, which we achieve mostly by storing the IR version in the `.sjsir` files. This strategy has allowed Scala.js 0.6.x to stay backward binary compatible for more than three years and a half at the time of writing, despite significant changes to the IR across versions (e.g., the introduction of non-native JS classes).

3.1 Overview

As an introduction to reading snippets of Scala.js IR, we will run through the IR produced for the following small Scala.js program:

```scala
package hello

import scala.scalajs.js

object HelloWorld {
  def main(args: Array[String]): Unit = {
    val names = List("Sébastien", "Antoine", "Sophie", "Alice")
    for (name <- names)
      js.Dynamic.global.console.log(greeting(name))
  }

  def greeting(name: String): String =
    "Hello " + name
}
```

The corresponding (unique) .sjsir file coming out of it would contain:

---

[1]https://github.com/lionelfleury/Kotlin-Scala.js

```
module class Lhello_HelloWorld$ extends O {
  def main__AT__V(args: T[]) {
    val names: sci_List = mod:sci_List$.apply__sc_Seq__sci_List(
      new sjs_js_WrappedArray().init___sjs_js_Array(
        ["S\u00e9bastien", "Antoine", "Sophie", "Alice"]));
    names.foreach__F1__V(
      new sjsr_AnonFunction1().init___sjs_js_Function1(
        (arrow-lambda<$this: Lhello_HelloWorld$ = this>(name$2: any) = {
          val name: T = name$2.asInstanceOf[T];
          $this.$$anonfun$main$1__p1__T__sjs_js_Dynamic(name)
        })))
  }
  def greeting__T__T(name: T): T = {
    ("Hello " +[string] name)
  }
  def $$anonfun$main$1__p1__T__sjs_js_Dynamic(name: T): any = {
    global:console["log"](
      mod:sjs_js_Any$.fromString__T__sjs_js_Any(
        mod:Lhello_HelloWorld$.greeting__T__T(name)))
  }
  def init___() {
    this.O::init___();
    mod:Lhello_HelloWorld$<-this
  }
}
```

Let us pick apart relevant pieces of that IR.

```
module class Lhello_HelloWorld$ extends O {
  ...
}
```

declares a *module class* (i.e., the class of a Scala `object`) named `Lhello_HelloWorld$` (the IR-encoded name of `hello.HelloWorld$`. The class extends `java.lang.Object` (whose IR name is `O`) and does not implement any interface.

The last method,

```
def init___() {
  this.O::init___();
  mod:Lhello_HelloWorld$<-this
}
```

is a *constructor,* as identified by its name starting with `init___`. It first calls the super parameter-less constructor, inherited from `Object`, then stores `this` as the singleton module instance of `HelloWorld$`.

The most interesting method is obviously `main`. Its signature,

```
def main__AT__V(args: T[]) {
```

indicates that it takes an array of `java.lang.String` (whose IR name is `T` for Text) and returns

void. Its mangled name repeats that information, with `AT` standing for array of `T`, and `V` for **void**. In general, the method names always reflect their signature, which is how we encode away overloading at the IR level.

The first statement declares a local immutable variable:

```
val names: sci_List = mod:sci_List$.apply__sc_Seq__sci_List(
  new sjs_js_WrappedArray().init___sjs_js_Array(
    ["S\u00e9bastien", "Antoine", "Sophie", "Alice"]));
```

On the last line, the `[a, b, c]` is a JavaScript array initializer. That JavaScript array is wrapped inside a `js.WrappedArray` to conform to `scala.collection.Seq`, which is the IR type of Scala varargs. Those varargs are given to the `apply` method of the singleton module instance of `scala.collection.immutable.List`, which builds a `List` out of varargs. That list is stored in the local `val names`, whose static type is a `sci_List`.

The second instruction is the for comprehension, which has been compiled as a call to the `foreach` method of `List`:

```
names.foreach__F1__V(
  new sjsr_AnonFunction1().init___sjs_js_Function1(
    (arrow-lambda<$this: Lhello_HelloWorld$ = this>(name$2: any) = {
      val name: T = name$2.asInstanceOf[T];
      $this.$$anonfun$main$1__p1__T__sjs_js_Dynamic(name)
    })));
```

The construct named **arrow-lambda** creates a new JavaScript arrow function, with one actual argument `name$2: any`. The part in `<...>` explicitly lists the *captures* of the arrow function. It reads as: capture the value of **this** in the surrounding scope, as the value `$this` of type `hello.HelloWorld$` inside the lambda. Captures are always immutable in the Scala.js IR.

Note that even though `name` was statically known to be a `String`, the parameter `name$2` of the lambda is of type **any**. It is cast inside the body of the lambda back to a `String`. This is a consequence of type erasure in Scala, which transfers to Scala.js: the generic parameters of `js.Function1[String, Unit]` are lost during erasure, and replaced with `Any`, leaving casts behind to fix the type of the local variable.

The JavaScript function produced by the **arrow-lambda** is wrapped inside an instance of `scala.scalajs.runtime.AnonFunction1` so that it conforms to `scala.Function1`, the type of unary Scala functions. That `AnonFunction1` is given to `foreach__F1__V`, a method which takes a `scala.Function1` (whose IR name is `F1`) and returns **void**.

Inside the body of the lambda, we call a compiler-generated private function:

```
def $$anonfun$main$1__p1__T__sjs_js_Dynamic(name: T): any = {
  global:console["log"](
    mod:sjs_js_Any$.fromString__T__sjs_js_Any(
      mod:Lhello_HelloWorld$.greeting__T__T(name)))
}
```

That method looks up the identifier `console` in the JavaScript global scope (`global:console`), then calls the JavaScript method `log` on it with arguments. JavaScript method calls, as all other JavaScript operations in the IR, are dynamically typed: they take `any`'s as parameters and return `any`'s. In general, all interactions with JavaScript code are typed as `any`, given that JavaScript code could do arbitrary stuff and return arbitrary values. Without that restriction, the IR would be decidedly unsound. It is the responsibility of the compiler from Scala.js to the IR to insert appropriate casts if necessary, in order to "protect its borders", as was discussed in Section 2.2.4. Peculiarities of Scala.js' encoding of `js.Dynamic` cause the call to `fromString__T__sjs_js_Any`, which is basically an identity function. We also call the method `greeting__T__T` defined as

```
def greeting__T__T(name: T): T = {
  ("Hello " +[string] name)
}
```

That function takes a `String` and returns a `String`. Note that `return` is implicit, as in Scala, since the IR is expression-based.

3.2 Definition

Now that we have an overview of how to read snippets of the IR, we can proceed to defining its structure. We define the IR syntax using a notation that is largely inspired by that of Featherweight Java [32], aka FJ, and its extension FeatherTrait Java with Interfaces [38], aka iFTJ. The full syntax and typing rules can be found in Appendix A. In this section, we will only point to some noteworthy pieces of it.

The run-time semantics, expressed as an extension to the ECMAScript specification, can be read online at http://sebastien.doeraene.be/thesis/sjsir-semantics/. They are unfortunately too long to put in print as an integral part of this document, and too difficult to navigate without hyperlinks anyway.

3.2.1 Class definitions

At a high level, an IR program is a pair (`CT`, `t`) of a class table and a *main expression*. The class table is simply a set of class definitions `CD`, of the form

```
[ CC ] CK C [ extends D [ via t ] ] [ implements I ] [ NLS ] { CM }
```

where `CC` are class-level captures (see JS class creation in Section 3.2.6), `CK` is the class kind, `C` is the class name, `D` and $\overline{\mathtt{I}}$ are superclass and implemented interfaces, `NLS` is the "native load spec" (used by the term `constructorOf[C]`) and $\overline{\mathtt{CM}}$ are the class members.

Unlike FJ which has only one kind of class definition, or even iFTJ which has classes and interfaces, the Scala.js has a total of 8 kinds of class definitions. The *class kind* (CK) can take one of the following values:

- `class`: a regular class, similar to iFTJ's class;

- `module class`: the class of a singleton `object`, very similar to a `class` but it has an implicit singleton instance, and cannot be otherwise instantiated nor extended;

- `interface`: an interface, similar to Java 8's `interface` and, to a lesser extent, to iFTJ's trait;

- `js class`: a JavaScript class definition (in Scala.js, one that is a subtype of `js.Any`);

- `module js class`: the class of a singleton JavaScript `object` (in Scala.js, one that is a subtype of `js.Any`);

- `abstract js type`: a completely abstract JavaScript type, such as those represented by `traits` extending `js.Any` in Scala.js;

- `native js class`: a `js class` that is not actually implemented in the IR, but is rather provided by some native JavaScript code (in Scala.js, a `class` with the `@js.native` annotation);

- `native js module class`: a `js module class` provided by some native JavaScript code (in Scala.js, an `object` with the `@js.native` annotation).

The first three kinds of classes are called *Scala classes*, and obey semantics that are JVM-like. Each such class implicitly defines a *type* with the same name. Operations on Scala classes, such as method calls, are statically typechecked. Instances of such classes are technically *exotic objects*, in the vocabulary of the ECMAScript specification, and are defined such that JavaScript code cannot do much with them. They can have explicit *exports* to open up some of their accesses to JavaScript code.

All other kinds of classes are called *JavaScript classes*, and obey JavaScript semantics. JavaScript classes do *not* define an associated type, i.e., the name of a JavaScript class cannot be used as valid type in the IR. All values of such classes are typed as `any` instead (even if they had a more precise type in the Scala.js source code), and operations on them are dynamically typed. JS classes and module JS classes are defined in the IR, whereas native ones only have a shallow facade in the IR and are actually defined elsewhere (typically in native JavaScript code). Abstract JS types are completely abstract shells in the IR, and only serve the purpose of defining the class at the IR level so that it can be used in array types.

The predicate istype(C) indicates whether C is a Scala class, i.e., whether it defines a valid type.

Similarly to the hard separation between Scala classes and JavaScript classes in the Scala.js class hierarchy, Scala classes and JavaScript classes cannot extend each other in the IR: a Scala class cannot extend or implement JavaScript classes, nor vice versa. There is one exception for technical purposes: native and abstract JavaScript classes can extend the Scala class `Object`.

The following rules apply to class definitions, although we do not spell them out as formal rules in Appendix A because of the combinatorial explosion of required rules (due to how we write the syntax of optional superclass and optional list of implemented interfaces):

- The superclass `D` of a `class` or `module class` must be a `class`, except for `Object` which has no super class. Its interfaces $\overline{I}$ must be of kind `interface`.

- An `interface` cannot have a superclass, and its implemented interfaces must be of kind `interface`.

- A `js class` or a `module js class` must have a superclass of kind `native js class` or `js class`.

- Native JS classes and native module JS classes must have a superclass of kind `js class` or `native js class`, or the special class `Object`.

- Abstract JS types can optionally have a superclass of kind `js class` or `native js class`.

- The implemented interfaces of all JS classes must be of kind `abstract js type`.

The `extends` and `implements` clause define the *subclass* relationship, noted `C < D`. It is defined as follows:

$$C < C \quad \text{(C-Refl)} \qquad C < \mathtt{Object} \ \text{(C-Obj)} \qquad \frac{C < D \quad D < E}{C < E} \\ \text{(C-Trans)}$$

$$\frac{[\ \overline{CC}\ ]\ \mathtt{CK}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ [\ \mathtt{via}\ \mathtt{t}\ ]\ [\ \mathtt{implements}\ \overline{I}\ ]\ [\ \mathtt{NLS}\ ]\ \{\ \overline{CM}\ \}}{C < D}\ \text{(C-Extends)}$$

$$\frac{I \in \overline{I} \quad [\ \overline{CC}\ ]\ \mathtt{CK}\ \mathtt{C}\ [\ \mathtt{extends}\ \mathtt{D}\ [\ \mathtt{via}\ \mathtt{t}\ ]\ ]\ \mathtt{implements}\ \overline{I}\ [\ \mathtt{NLS}\ ]\ \{\ \overline{CM}\ \}}{C < I} \\ \text{(C-Implements)}$$

The subclass relationship is distinct from the *subtyping* relationship, written `S <: T`, which we show in the next section. In particular, `<` applies to JavaScript classes as well as Scala classes, although JavaScript classes do not define an associated type.

Similarly to FJ, we assume that the class table is such that there are no cycles in the subclass relationship, i.e., that `<` is antisymmetric. We also stress that a class `C` cannot directly extend or implement itself, either.[2] Unlike in FJ, the class `Object` is assumed to be part of the class table. It must not have any superclass nor implement any interface, but otherwise obeys the same rules as all the other classes. We also assume that no class defines two members of the same category with the same name (e.g., two instance fields with the same name, or two static methods with the same name). A practical implementation must check those sanity rules

---

[2]The authors of FJ do not explicitly forbid this, although it was obviously intended.

early, before attempting to apply any other helper functions or typing judgment defined in the specification.

3.2.2 Types

As we already mentioned in the previous section, in the IR, every Scala class (of kind `class`, `module class` or `interface`) implicitly defines a *type* of the same name. In addition, the IR features primitive types, four special types `void`, `any`, `nothing` and `null`, and array types. The full syntax of types is as follows:

```
 T  :=  void | any | nothing | null | PT | C | ETR[]
PT  :=  boolean | char | byte | short | int | long | float | double
     |  string | undef
```

where `ETR[]` are array types, and are further defined in Section A.1.1. While syntactically, any class name `C` can be used as a type `T`, not every class name is a *valid type*. Only class names referring to a Scala class are valid types. Invalid types must be excluded in some typechecking rules. `string` and `undef` are the types of JavaScript primitive `string`s and of the `undefined` value, respectively. The other primitive types are equivalent to those of the JVM, among which `boolean` and `double` which coincide with JavaScript primitive `boolean`s and `number`s, respectively. `null` is the type of the JavaScript value `null`, and therefore looks very much like a primitive type. However, it has a special role in our type system, since it is a subtype of all reference types (class types and array types)—unlike `undef`.

`void` is the type given to *statements*, i.e., terms that do not evaluate to a value (in the run-time semantics, they return a completion record with value `empty`). Note that it is different from the typical `unit` type in that it is not a subtype of `any`. In fact, because any expression can be used in statement position (the value it evaluates to being discarded), our type system contains the unusual rule `T <: void` for any `T`, including `any`. `any` is therefore not technically the top type, although it is a supertype of all the other types. `nothing` is a standard bottom type.

The full (algorithmic) subtyping rules can be seen in Section A.2.3. Most rules are fairly standard, once we have established that `void` is the real top type and that only Scala classes (for which istype(C) is true) have an associated type. Rules for array types mimic those on the JVM: arrays are covariant (which means that array stores must validate their input with a cast) and also subclasses of `Object`, `java.lang.Cloneable` and `java.io.Serializable`.

The most unusual rule is S-PRIMCL:

$$\frac{\text{boxedcl(PT)} < \text{C}}{\text{PT} <: \text{C}} \qquad \text{(S-PrimCl)}$$

which says that each primitive type (e.g., `int`) is a subtype of its corresponding boxed class (for `int`, it is `java.lang.Integer`) and all its superclasses. In fact, in the IR, the boxed classes do not have any instance besides the corresponding primitive values. For example, a value of type `String` is either `null` or a primitive JavaScript `string`, but never an object.

3.2.3 Method Names and Type References

Whereas class names (of metavariables `C`, `D` and `E`) are uninterpreted identifiers, method names (and constructor names) have some structure in the IR. A method name, represented by the metavariable `m`, is internally of the form `x(TR)TR`, which encodes both the *base name* (an identifier `x`) as well as the *type references* for paramter types and result type. The syntax of type references, abbreviated as type-refs, is given in Section A.1.1. They differ from types in two ways:

- there is no type-ref corresponding to the types `any`, `string` and `undef`, and

- there are type-refs for JS class names in addition to Scala class names.

The precise structure of type-refs may seem obscure and arbitrary (why exclude `string` and `undef`, for example?). They are in fact direct equivalents of JVM types, which are used for the encoding of overloading. On the JVM, a method identity comprises its base name and its signature, and methods are always referred to by their full identity. This mechanism encodes overloading away, effectively making the JVM language overloading-free.

The IR uses exactly the same mechanism. The signature part of a method name uses type-refs, instead of types, so that overloading for Scala classes in Scala.js perfectly matches overloading in Scala/JVM. Whereas on the JVM, the signature directly correlates with the types of parameters and result type, in the IR we need a transformation to account for the differences between type-refs and types. The function mtype(`m`) computes the type signature of a method name.

Constructor names (metavariable `k`) are similar to method names, except that their base name is always `<init>` and they do not have any result type. They have their own function ktype(`k`) to compute type signatures.

$$\text{trtp}(C) = \text{clstp}(C) \quad (\text{TRTP-CLS}) \qquad \frac{\text{TR} \neq C}{\text{trtp}(\text{TR}) = \text{TR}} \; (\text{TRTP-NONCLS})$$

$$\text{mtype}(x(\overline{\text{TR}})\,\text{TR1}) = (\text{trtp}(\overline{\text{TR}}))\,\text{trtp}(\text{TR1}) \qquad \text{ktype}(\texttt{<init>}(\overline{\text{TR}})) = (\text{trtp}(\overline{\text{TR}}))$$
$$(\text{METH-TYPE}) \qquad\qquad\qquad\qquad\qquad (\text{CTOR-TYPE})$$

There is a third kind of name, namely reflective proxy names (metavariable `p`), together with its ptype(`p`) function, which are exclusively used by reflective calls (see Section 3.2.6).

Method names and constructor names found in the program are assumed to be well-formed: they must reference only existing class names, and parameter type-refs cannot be `void`.

3.2.4   Typing class members

Class definitions are typed by typing all their members. Before looking at their types, we give a very brief intuition on what each kind of member represents. In general, members whose name is an identifier, a method name or a constructor name (as defined in the previous section) are never accessible from JavaScript code. Only members whose name is [t] for some expression t can be seen from JavaScript, as well as top-level exports.

- `static (val|var) x: T`
  A static field of type `T`, immutable (`val`) or mutable (`var`).

- `static def m(`$\overline{\text{[var] x: T}}$`): T = t`
  A static method. As anywhere else, parameters are optionally mutable.

- `(val|var) x: T`
  An instance field of type `T`.

- `def k(`$\overline{\text{[var] x: T}}$`) { t }`
  A constructor in a Scala `class` or `module class`. Constructors can mutate even immutable fields of `this`, for purposes of initialization. JS classes do not have a *constructor* as an IR concept (they can have a JS method whose name happens to be the constant string `"constructor"`).

- `def m(`$\overline{\text{[var] x: T}}$`): T = t`
  A concrete method in a Scala `class`, `module class` or `interface`. In an interface, it behaves like a `default` method from Java 8+.

- `def m(`$\overline{\text{x: T}}$`): T`
  An abstract method in any kind of Scala class.

- `[ static ] (val|var) [t]: T`
  A field in a `js class` or `js module class`, static or not.

- `[ static ] def [t](`$\overline{\text{[var] x: any}}$` [ [var] ...x: any ]): any = t`
  A method visible from JavaScript, static or not. In a Scala `class` or `module class`, it cannot be static, and t must be a string literal.

- `[ static ] prop [t] [ get() = t ] [ set([var] x: any) { t } ]`
  A property definition, visible from JavaScript, static or not. Either the getter, the setter, or both, must be present. In a Scala `class` or `module class`, it cannot be static, and t must be a string literal.

- `export top class <string literal>`
  A top-level export of the class value corresponding to a top-level `js class`.

- `export top module <string literal>`
  A top-level export of the singleton instance of a `module class` or `js module class`.

- `export top static def <string literal>(`
  $\overline{\texttt{[var] x: any}}$ `[ [var] ...x: any ]): any = t`
  An exported top-level `function` definition.

- `export top static field x as <string literal>`
  A read-only top-level export of a static field. The field must have type `any`.

Since member typings depend on the enclosing class (e.g., the type of `this`), the typing judgments for class members follow the shape

```
CM OK IN C
```

meaning that the member `CM` is allowed and well-typed if found within the class `C`. This is similar to the notation of FJ and iFTJ. Typing rules for members can be found in Section A.2.4. Most rules depend on the kind of the enclosing class, which is looked up with clskind(C).

Consider the rule METH-OK for statically typed concrete instance methods.

$$\frac{\text{clskind(C)} \in \{\texttt{class}, \texttt{module class}, \texttt{interface}\} \quad}{\texttt{mtype(m)} = (\overline{\texttt{T}})\texttt{T1} \quad \varnothing, \texttt{this:C}, \overline{\texttt{[var]x:T}} \vdash \texttt{t:S1} \quad \texttt{S1 <: T1}}{\texttt{def m(\overline{[var] x: T}): T1 = t OK IN C}} \text{(METH-OK)}$$

The first premise demands that the enclosing class be a `class`, `module class` or `interface`, i.e., a Scala class. Indeed, JavaScript classes cannot have any statically typed instance methods. The second premise asserts that the parameter types and result type match what is predicted by the method name, i.e., mtype(m). Finally, the last two premises check that the body of the method can be typed with an empty environment $\varnothing$ augmented with bindings for `this` and the formal parameters, and that the result type is a subtype of the declared result type (the type system does not have subsumption).

Most member typing rules have some subtlety or another. We only present a few of them.

- The rule FIELD-OK for statically typed instance fields forbids two things: fields cannot have type `void`, and no field is allowed in any of the ancestors of boxed classes (including themselves). The latter is necessary because boxed classes are not represented as objects, but rather as their associated primitive type. Since primitive values cannot receive additional fields in JavaScript, this must be forbidden at the IR level.

- The rule CTOR-OK for constructors puts the special token $\mathscr{C}(\texttt{C})$ in the environment, which indicates that we are in the body of a constructor of `C`. This will later be used to allow the initialization of immutable fields from the constructors (rule T-ASSVALFIELD) but not from anywhere else.

- The four rules for JS members (JSFIELD-OK, JSCTOR-OK, JSMETH-OK and JSPROP-OK) take the *class environment* of C, clsenv(C), into account. The class environment contains bindings for class captures, if any. This class environment can be used both in the body of methods, and for the very *name* of the member. Indeed, in JavaScript, members can have *computed names*, which are computed at class creation time and can depend on the lexical environment. Classic constant-string-named members are represented with the same syntax, using a string literal as the member name.

- The rule for JSCTOR-OK expects a specific shape of body, with exactly one super constructor call (there is no term typing rule for JS super constructor calls, so this is the only valid location for them). In those constructors, the `this` value is part of the environment only *after* the super constructor call statement.

It is worth noting that all members that are visible by JavaScript code—whether exported members in Scala classes or JS members in JS classes—are fully typed with `any`s. For example, a method takes parameters of type `any` and return values of type `any`. Since JavaScript is dynamically typed and out of our control, we cannot force the JS code to abide by more precise types. When the Scala.js compiler compiles `@JSExport`'ed members down to the IR, it inserts the necessary casts from `any` down to the expected parameter types.

An alternative would have been to allow more precise types in the IR and include the implicit casts in the run-time semantics of the IR, which would be closer to the tradition of gradual typing. This would have simplified the compiler's job, but would have required to deal with run-time overloading dispatch in the IR specification as well, significantly complicating optimizations. With the existing specification, run-time overloading dispatch is taken care of by the compiler (similarly to the compile-time overloading semantics of Scala classes), which simplifies the IR semantics and makes reasoning about the IR easier.

3.2.5 Typing class definitions

Now that we know how to type class members, we finally look at the typing rules for class definitions. Essentially, class definitions are valid only if all their members are well-typed, i.e.,

$$\frac{\overline{\text{CM}} \text{ OK IN C}}{[\ \overline{\text{CC}}\ ] \text{ CK C } [\text{ extends D } [\text{ via t }]\ ] [\text{ implements } \overline{\text{I}}\ ] [\text{ NLS }] \{\ \overline{\text{CM}}\ \} \text{ OK}}$$

The actual rules, shown in Section A.2.4, also check that a class does not declare the same fields as its parent classes.

3.2.6 Typing terms

So far, we have seen high-level properties of classes, and typing rules for class definitions and their members. Once we "enter" a method body, however, we have to look at typing rules for terms. We use a standard typing relation of the form

```
Γ ⊢ t : T
```

to indicate that the term `t` has type `T` under the environment Γ. The latter is a sequence of assumptions, each of one of the following forms:

- `this:T` says the type assumed for the special value `this` in Γ is `T`,

- `[var]x:T` says that the type assumed for the local variable `x` in Γ is `T`, and it is mutable if and only if the `var` token is present,

- $\mathscr{C}$`(C)` says that we assume that the term is located inside a constructor of the class `C`, and

- α`:T` says that the result type associated with the label α in Γ is assumed to be `T`.

We use ∅ to denote the empty environment, and a comma to concatenate additional elements to an environment.

The typing rules for terms can be found in Section A.2.5. They are algorithmic, thanks to the following observations:

- typing rules have distinct syntactical terms in their conclusion, with a few exceptions where the premises contain clearly opposed predicates (e.g., T-AssFld and T-AssValFld can be distinguished based on the presence or absence of `var`),

- the type `T` of the term in the conclusion is always a function of a) the syntax (e.g., in T-If), b) the class table (e.g., in T-FldRef) and c) direct typing of sub-terms (e.g., in T-ValDef),

- the environments used when typing a subterm is always a function of a) the environment in the conclusion and b) the syntax.

This means that we can typecheck terms in two passes: first compute environments in a top-down traversal, then assign types to terms from bottom to top and check the premises.

The IR unfortunately defines a lot of different kinds of terms. This is caused by its main requirement that it be able to represent both the statically typed operations of Scala classes and *all* the operations that can be expressed in ECMAScript 2015. The latter part is a direct consequence of the completeness criterium that we elaborated in Section 2.2.6 for the Scala.js language. If Scala.js can express everything that ECMAScript 2015 can, and we compile Scala.js down the IR, it follows that the IR must be able to represent all of ECMAScript 2015 as well. The ability to represent a wide variety of statically typed operations for Scala classes is less crucial. We *could* have implemented the Scala parts of Scala.js in a dynamically typed IR by

desugaring more in the compiler. However, that would defeat the purpose of using an IR for optimizations, since most optimizations rely on its statically typed aspect.

Despite the large amount of terms, most of them are unsurprising. The `val` and `var` declarations are standard `let-in` constructs; the `if` control structure is standard as well; etc. We only highlight some special terms, as well as a few representatives of larger categories.

If condition

As an example of a simple term, consider the rule T-IF:

$$\frac{\begin{array}{ccc} \Gamma \vdash \texttt{t1:S1} & \Gamma \vdash \texttt{t2:S2} & \Gamma \vdash \texttt{t3:S3} \\ \texttt{S1 <: boolean} & \texttt{S2 <: T} & \texttt{S3 <: T} \end{array}}{\Gamma \vdash \texttt{if[T] (t1) t2 else t3 : T}} \tag{T-IF}$$

Besides the typical shape of a typing rule for an `if`, there are two things worth noting: First, the type of the `if` expression is explicitly stated in the syntax, as `if[T]`. This ensure that we do not need to compute the LUB of the types of `t2` and `t3`. Indeed, LUB's in a type system with interfaces and without union types are icky, and we are eager to avoid any such complexity in the IR. In a source language, having to write the result type of every `if` would be inconvenient to say the least, but in an IR it is perfectly fine.

The second observation is that we explicitly have pairs of premises such that $\Gamma \vdash \texttt{t1:S1}$ and `S1 <: boolean`. This is necessary because the typing rules do not have any built-in subsumption, to keep them syntax-directed. Note that `S1 <: boolean` is different from `S1 = boolean` because it also allows `S1 = nothing`.

Labeled block and return

JavaScript features an unusual control structure called the labeled statement, which takes the following shape:
```
label: {
  ...
  if (x)
    break label;
  ...
}
```

Within the `label: {...}` block, the instruction `break label;` is valid, and forces a jump to the statement located *after* the entire block. It is therefore a *jump out* of the labeled block.

In the IR, we have a generalized form of the labeled statement that can also be used as an expression. It takes the following shape:

```
val y: T = label[T]: {
  ...
  if (x)
    return@label someT;
  ...
  someOtherT
}
```

Similarly to an `if` term, the syntax of a labeled block specifies the result type of the expression. The labeled block is an expression of that type, and can therefore be used in a position where a term of type `T` is expected.

The body of the labeled block must typecheck as a term of type `T`. If execution of the body does not throw or return, the resulting value is used as the result of the labeled block. In addition, inside the body, the expression `return@label t` is allowed if `t` is of type `T`. If execution reaches such a `return`, the execution of the body is aborted, and the result of the labeled block as a whole is the result of evaluating `t`.

If we take `T = void`, the labeled block specializes to a labeled statement as in JavaScript (where `return` is used instead of `break`), which is why we consider the labeled block as a generalization of the labeled statement.

The typing rules for the labeled block and return are the following:

$$\frac{\Gamma, \alpha\texttt{:T} \vdash \texttt{t1:S} \quad\quad \texttt{S <: T}}{\Gamma \vdash (\alpha\texttt{[T]: \{ t1 \}): T}} \quad\quad \text{(T-LABELED)}$$

$$\frac{\Gamma \vdash \texttt{t1:S} \quad\quad \alpha\texttt{:T} \in \Gamma \quad\quad \texttt{S} \neq \texttt{void} \quad\quad \texttt{S <: T}}{\Gamma \vdash \texttt{return@}\alpha \texttt{ t1 : nothing}} \quad\quad \text{(T-RETURN)}$$

T-LABELED says that when typechecking the body `t1`, we have the pair $\alpha\texttt{:T}$ in the environment, to remember that there is an enclosing labeled block with label α and result type `T`. T-RETURN requires the appropriate pair to be found in the environment, and typechecks the expression accordingly.

The labeled block is an extremely versatile control structure. We can use it to encode a variety of well-known constructs. For example, consider the classical `return` out of the enclosing function, which we could write as

```
def foo__I__I(x: int): int = {
  if (x < 0) {
    return -x
  }
  x
}
```

Instead of having an implicit return point, which is the function body, we can use a labeled

block and a specific return as follows:

```
def foo__I__I(x: int): int = {
  ret[int]: {
    if (x < 0) {
      return@ret -x
    }
    x
  }
}
```

Indeed, if the execution does not encounter the `return`, the last expression of the block, x, is evaluated and returned. And if we execute the `return@ret -x`, we jump out of the block with the result -x. Since the block is immediately enclosed in the body of the function, the result of the block is always the result of the function itself.

Another, more interesting example, is to encode `break` and `continue` inside loops. Consider the following loop:

```
def foo__I__I(x: int): void = {
  var i: int = x
  while (i > 0) {
    if (i % 3 == 0)
      continue
    if (i % 10 == 0)
      break
    println(i)
    i = i + 1
  }
}
```

We can encode away `continue` and `break` with two labeled blocks, one outside the loop, and one inside the loop:

```
def foo__I__I(x: int): void = {
  var i: int = x
  breakLoop[void]: {
    while (i > 0) {
      continueLoop[void]: {
        if (i % 3 == 0)
          return@continueLoop undefined
        if (i % 10 == 0)
          return@breakLoop undefined
        println(i)
        i = i + 1
      }
    }
  }
}
```

The pattern also works for labeled loops with their `continue label` and `break label` statements.

When compiling Scala.js source code to the IR, the compiler also uses labeled blocks in the translation of pattern matches, whereas the JVM compiler uses unstructured control flow. They are also used to compile tail-recursive calls. For example, the following source code:

```
def foldLeft[A, B](xs: List[A], z: B)(f: (B, A) => B): B =
  if (xs.isEmpty) z
  else foldLeft(xs.tail, f(z, xs.head))(f)
```

compiles down to the following IR:

```
def foldLeft__sci_List__O__F2__O(
    var xs: sci_List, var z: any, f: F2): any = {
  x[any]: {
    while (true) {
      _foldLeft: {
        return@x {
          if (xs.isEmpty__Z()) {
            z
          } else {
            val temp$xs: sci_List = xs.tail__O().asInstanceOf[sci_List];
            val temp$z: any = f.apply__O__O__O(z, xs.head__O());
            xs = temp$xs;
            z = temp$z;
            return@_foldLeft (void 0)
          }
        }
      }
    }
  }
}
```

There are several interesting things to note in the above snippet:

- The `x[any]` label at the top of the function is the encoding of `return` from an entire function, which we saw before.

- The `while` loop is a valid body for that label because its type is `nothing`, according to T-WHILETRUE.

- The inner `_foldLeft` label is the encoding of a `continue`.

- The `return@x { t }` bypasses the loop to return `t` (assuming `t` evaluates to a value), which encodes the normal control flow out of the tail-recursive loop, in case it *doesn't* call itself (this happens in the empty list case).

- The `return@_foldLeft` bypasses *that* bypass, to force the looping, which is how the tail-recursive call itself is compiled.

Having a unique control structure instead of these various constructs significantly simplifies the IR and its handling. In particular, the labeled blocks and their returns are the only structure that manipulate labels, unlike in JavaScript where they are used in loops and switches in addition to the labeled statement itself, with various interactions with label-less `break`s and

`continues`, etc. The absence of function-return statement also simplifies inlining, in a context where our IR is tree-based, since we can simply splice the target body at the call site (with appropriate substitution).

Although it seems like the IR for a tail-recursive loop is too "ugly" as a result, the very last transformation phase, from the IR to JavaScript code, can simplify typical patterns. As it turns out, the above `foldLeft` gets compiled to the following JavaScript code:

```
foldLeft__sci_List__O__F2__O(xs, z, f) {
  while (true) {
    if (xs.isEmpty__Z()) {
      return z
    } else {
      const this$1 = xs;
      const temp$xs = this$1.tail__sci_List();
      const temp$z = f.apply__O__O__O(z, xs.head__O());
      xs = temp$xs;
      z = temp$z
    }
  }
};
```

Instance method application

Among the statically typed operations, perhaps the best representative is instance method application, whose syntax is

$$\texttt{t1.m(}\overline{\texttt{t}}\texttt{)}$$

Intuitively, its run-time semantics are:

1. Evaluate `t1` as `v1`

2. If `v1` is `null`, trigger an Undefined Behavior of kind `NullPointerException`

3. Evaluate each element of $\overline{\texttt{t}}$ as $\overline{\texttt{v}}$, respectively

4. Find the body of `m` in the class of `v1`, and execute it, substituting `v1` for `this` and $\overline{\texttt{v}}$ for the formal parameters.

The main typing rule for method application is T-APPLY (there is an additional rule for the case where the receiver has type `nothing` or `null`, namely T-APPLYNULL):

$$\frac{\begin{array}{cc} \texttt{m} \in \text{methods}(\text{tpcls}(\texttt{T1})) & \text{mtype}(\texttt{m}) = (\overline{\texttt{T}})\texttt{R} \\ \Gamma \vdash \texttt{t1:T1} \quad \texttt{T1} \notin \{\texttt{void},\texttt{null},\texttt{nothing}\} \quad \Gamma \vdash \overline{\texttt{t}}\texttt{:}\overline{\texttt{S}} \quad \overline{\texttt{S}} \texttt{ <: } \overline{\texttt{T}} \end{array}}{\Gamma \vdash \texttt{t1.m(}\overline{\texttt{t}}\texttt{)} \texttt{ : R}} \quad \text{(T-APPLY)}$$

When T1 is a class name of the form `C1`, it is clear that we have to verify the existence of `m` in the methods of `C1`, i.e., `m` ∈ methods(`C1`). However, the above typing rule allows `t1`'s type to

be any type but `null` and `nothing`. In particular, it can be a primitive type, such as `int`. This is at odds with typical systems mixing objects and primitive values, where method calls are forbidden on values of primitive types. Recall, however, that primitive values are the actual instances of their corresponding boxed classes. Given that, we must be allowed to call methods on primitive values, in which case the method to call must be the one found in the appropriate boxed class.

This is why we use the helper function tpcls(T1) to find the appropriate "implementation class" of a given type. Besides primitive types, tpcls() can also resolve array types (to the `Object` class[3]) and, perhaps surprisingly, the `any` type (also to `Object`). This means that we can call any method of `Object` even on JavaScript objects!

Clearly, the run-time semantics must also give meaning to such calls, despite the fact that JavaScript objects and primitive values obviously cannot actually implement the methods of `Object` and the boxed classes. The rule is that, when the receiver evaluates to a non-Scala object, its run-time type is used to derive its implementation class (e.g., a `number` maps to `Double`) and the appropriate method in the implementation class is called, with the receiver as `this` value (this also means that the `this` value inside methods of ancestors of boxed classes may be a primitive!). There is one exception: for a JavaScript object, calling `x.toString__T()` resolves into calling the JavaScript method `x.toString()`.

JavaScript method application

By opposition to statically typed method application, which we discussed in the previous section, we take a look at JavaScript method application, whose typing rule is

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: any} \quad \Gamma \vdash \overline{\texttt{t}}\texttt{:}\overline{\texttt{S}} \quad \overline{\texttt{S}} \texttt{ <: any}}{\Gamma \vdash \texttt{t1[t2](}\overline{\texttt{[...]t}}\texttt{) : any}}$$

$$\text{(T-JSMETHAPPLY)}$$

It only demands that `t1`, `t2` and all $\overline{\texttt{t}}$ be valid expressions of type `any`. Note that `S <: any` is equivalent to $\texttt{S} \neq \texttt{void}$, so it requires that none of the receiver, method property name and parameters are statements.

The run-time semantics intuitively correspond to those of a JavaScript method call, more precisely a *CoverCallExpressionAndAsyncArrowHead* whose *MemberExpression* child is of the form *MemberExpression* [*Expression*], as found in Section 12.3.4.1 of the ECMAScript specification. In less obscure terms, it corresponds to the JavaScript construct
```
t1[t2](arg1, ..., argN)
```

---

[3]In practice, array classes have a custom implementation of `clone__O`, but for typechecking we do not need to take it into account, since it is assumed to be defined by `Object` anyway.

The operation is dynamically typed, as obvious from the `any` requirements everywhere. It is valid to call any method on any kind of receiver, with any number of arguments of any type. If the receiver does not support the given method, a `TypeError` will be thrown, as in the run-time semantics of *CoverCallExpressionAndAsyncArrowHead*.

In particular, the receiver can very well be a Scala object, optionally typed as such. In that case, the call is unlikely to be valid, unless t2 evaluates to a `string` v2, and the Scala object *exports* a method with name v2.

Other kinds of expressions for JS operations follow a similar scheme, where all subterms need to typecheck as subtypes of `any`, and the semantics are equivalent to the corresponding constructs of the ECMAScript Language. The constructs with `super` keywords are slightly different than their ECMAScript equivalent, in that they explicitly take a reference to the superclass, whereas in ECMAScript the superclass is inferred from the context. The variant found in the IR makes the semantics of the IR construct independent of its lexical context, which allows to more easily inline methods containing such nodes.

Closure creation

In JavaScript, there are two ways to create function objects: arrow functions and "function" functions. Those two kinds of functions have some distinct run-time semantics, which means we need our IR to be able to create both. The syntax for an arrow function creation is

$$\texttt{arrow-lambda<}\overline{\texttt{x1:T1=t1}}\texttt{>(}\overline{\texttt{[var] x2: any}}\texttt{ [[var] ... x3: any]) = t}$$

which enforces that only the last parameter can be a rest parameter, if any. We have already introduced the intuition for lambda creation in Section 3.1. In particular, recall that captures are explicit in the syntax, and always immutable. The terms $\overline{\texttt{t1}}$ are evaluated once at closure creation time, and the values are stored in the run-time environment of the lambda as $\overline{\texttt{x1}}$. The typing rule for arrow lambdas is T-ARROWLAMBDA. It merges the optional rest parameter together with the others to simplify the typing, since rest parameters do not influence typing, and the syntax itself prevents ill-formed rest parameters:

$$\frac{\overline{\texttt{T1}} \neq \texttt{void} \quad \Gamma \vdash \overline{\texttt{t1}} : \overline{\texttt{S1}} \quad \overline{\texttt{S1}} \texttt{ <: } \overline{\texttt{T1}} \quad \varnothing, \overline{\texttt{x1}} : \overline{\texttt{T1}}, \overline{\texttt{x2}} : \texttt{any} \vdash \texttt{t} : \texttt{any}}{\Gamma \vdash \texttt{arrow-lambda<}\overline{\texttt{x1:T1=t1}}\texttt{>(}\overline{\texttt{[var] [...] x2: any}}\texttt{) = t : any}} \text{(T-ARROWLAMBDA)}$$

In JavaScript, an arrow lambda does not introduce a binding for `this`. Instead, occurrences of `this` refer to the `this` of the enclosing scope. Consequently, an IR `arrow-lambda` does not introduce a binding for `this` in the body's environment. However, it does not inherit the binding for `this` from the enclosing environment Γ either. If the body needs to access the enclosing `this` value, it needs to be explicitly captured (as an identifier, since `this` is not syntactically valid in the definitions of captures).

Intuitively, the run-time semantics of an `arrow-lambda` correspond to two nested arrow functions in JavaScript:

```
((x11, ..., x1N) => {
  return ((x21, ..., x2M) => {
    return t;
  });
})(t11, ..., t1N)
```

the outer lambda being immediately applied, and ensuring that the $\overline{\text{t1}}$ are evaluated once upon lambda creation.

The rule for function lambdas, T-FUNCTIONLAMBDA, is similar, but does introduce a binding for `this`.

Since the IR does not define a way to access the metaproperty `new.target` yet, neither rule mentions it. The absence of support of `new.target` is, to the best of our knowledge, the only missing piece in the IR with respect to ECMAScript 2015.

JS class creation

In order to create a new class value at run-time, possibly with captures, the IR features the JS class creation expression, whose syntax is

$$\texttt{createjsclass[C]}(\overline{\text{t1}})$$

The class name C must refer to a non-top-level `js class` definition, of the form

$$\overline{\text{x1:T1}} \text{ js class C [extends C1 [via t]] [implements } \overline{\text{C2}} \text{] \{ } \overline{\text{CM}} \text{ \}}$$

The actual parameters $\overline{\text{t1}}$ are evaluated at class creation time, and stored as the class captures $\overline{\text{x1}}$, similarly to captures in lambdas. The typing rule is T-CREATEJSCLASS:

$$
\frac{\overline{\text{x1:T1}} \text{ js class C [extends C1 [via t]] [implements } \overline{\text{C2}} \text{] \{ } \overline{\text{CM}} \text{ \}} \qquad \Gamma \vdash \overline{\text{t1:S1}} \qquad \overline{\text{S1}} <: \overline{\text{T1}}}{\Gamma \vdash \texttt{createjsclass[C]}(\overline{\text{t1}}) : \text{any}}
$$

<div align="right">(T-CREATEJSCLASS)</div>

Every time the `createjsclass` expression is executed, it returns a new class value based on the template defined by C. This is true even if the class capture list $\overline{\text{x1:T1}}$ is empty (but present).

If the superclass reference t, or any other member declaration within $\overline{\text{CM}}$, needs to refer to variables in the lexical scope of the `createjsclass` invocation, they must be made available in the class captures.

This IR construct is the only one that allows the creation of arbitrary new JS class values at

run-time. It cannot be replaced by `function-lambda` expressions and imperative patches to the `prototype` because JS `classes` have unique semantic capabilities that `function-lambda` lack.

Reflective calls

Recall from Section 2.1.4 that Scala features structural types. Together with arbitrary casts, they basically allow to call arbitrary methods on arbitrary objects, in a reflective way. On the JVM, the compiler generates significant amounts of code using the Java reflection API to call the appropriate methods. Since Scala.js does not have reflection, we need an entirely different encoding of so-called reflective calls.

It turns out that in order not to destroy any attempts at dead code elimination (which is critical in the context of JavaScript, for code size), it is necessary to provision for this specific feature in the IR. As far as typing rules are concerned, fortunately, it is not too problematic. A reflective call has the following syntax:

`t1.p(`$\overline{\mathtt{t}}$`)`

and its typing rule is relatively simple:

$$\frac{\mathrm{ptype(p)} = (\overline{\mathtt{T}})\mathtt{any} \qquad \Gamma \vdash \mathtt{t1:S1} \qquad \mathtt{S1 <: any} \qquad \Gamma \vdash \overline{\mathtt{t}}:\overline{\mathtt{S}} \qquad \overline{\mathtt{S}} \mathtt{<:} \ \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{t1.p(}\overline{\mathtt{t}}\mathtt{)} \ : \ \mathtt{any}} \quad \text{(T-ApplyRefl)}$$

It says that, given any receiver, we can call any reflective proxy (of name `p`) with appropriate arguments. The result type is always `any`.

The run-time semantics of such a call are more complicated, however. Unlike regular method applications which have a full method name `m` or the form `x(`$\overline{\mathtt{TR}}$`)TR` to resolve, a reflective call only has a partial reflective proxy name `p` of the form `x(`$\overline{\mathtt{TR}}$`)`. More specifically, it lacks the result type. The semantics mimic the method resolution that is applied on the JVM using the Java reflection API. From the base method name and the list of parameter types, Scala/JVM uses the `java.lang.Class.getMethod()` API[4] to obtain a reference to the full method identity. The algorithm of `getMethod()` looks up the method in the target class, its superclasses, and eventually its superinterfaces (notably for default methods). In case two matching methods are found in the same class (with different result types), it selects the one with the most specific result type if it exists, or an arbitrary one.

The run-time semantics of a reflective call in the IR reproduces those steps, so that it selects the same method as would Scala/JVM. If it enters the last rule with selecting an arbitrary method,

---

[4]https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html

there is no best answer that would always be correct. But at the same time, it also means that the Scala/JVM-generated code is buggy for that case, as the wrong method can be called at run-time. We therefore preserve all correct behaviors, and are bug-compatible in the incorrect ones.[5] If no method is found, a `TypeError` is thrown (rather than a `NoSuchMethodException` as on the JVM).

In practice, a compiler from the IR to JavaScript can be much smarter than the naive semantics. Instead of resolving calls at run-time, it can precompute all possible targets, and insert special methods in the possible target classes as bridges, when linking. This mechanism is performed in collaboration with the reachability analysis and dead code elimination.

3.3 Properties of the IR

Unfortunately, proving any property of a language as rich as the IR is a huge undertaking, which we did not attempt. There are, however, a number of properties which we *expect* to hold. If those properties are shown not to hold, there is a bug in the specification, which would need to be fixed as soon as possible. Since the introduction of the IR in Scala.js 0.5.0, almost 4 years ago, we have discovered exactly one specification bug[6]—although we have regularly found bugs in the implementation of that specification, particularly in the type checker.

3.3.1 Type Soundness

Defining type soundness for the IR cannot follow standard practice, as our run-time semantics are not defined in terms of evaluation rules but rather in ECMAScript's Specese. Specese is basically an imperative, object-oriented metalanguage, where statements and expressions of the described language (Parse Nodes) live in a different world than values, meaning that even stating something like preservation using standard notation is problematic. Instead, we define type soundness for the IR as follows:

Definition 1 (Type Soundness for the IR)**.** The Scala.js IR is type-safe if and only if performing the abstract operation RunJobs() never runs into an *Assert* statement whose condition evaluates to **false**, assuming that each *sourceText* corresponding to a Scala.js IR program is backed by a well-typed IR program.

Proving Type Soundness for the Scala.js IR would be an immense endeavor, which we have not attempted in the context of this thesis. We conjecture that the current specification is type-safe, though, based on two main observations:

---

[5]It can be surprising and distressing to realize that portability sometimes requires explicit bug-compatibility.
[6]https://github.com/scala-js/scala-js/issues/3085

- the typing rules and subtyping relationship for statically typed operations are similar to the JVM verification rules, and

- every time we receive a value from JavaScript code, it is initially typed as `any`, which takes into account the fact that we do not know anything about what JavaScript code will do.

For future work, we foresee two approaches that could lead to proving type soundness. The first one would be to extend JSCert [5] with the Scala.js IR syntax, typing rules and evaluation semantics. This approach should be amenable to provide a complete proof that no *Assert* fails, including in the proof all the behavior that JavaScript code can do. The second approach would be to abstract away all of what JavaScript does, and take a different route centered on the specific semantics on the IR, unburdened by the semantics of all of JavaScript. This might be a more practical approach, and would also be more easily retargetable to other back-ends such as WebAssembly. That approach would however require to use some theoretical foundation to represent code that does "anything" (for some suitable definition of "anything") in the definitions and proofs.

3.3.2 Closed World

Besides type soundness, another very important property of the Scala.js IR is that it lives in a closed world. Now, that seems like an absurd notion, given that it is explicitly designed to interoperate with JavaScript code, on which we have no control and about which we have no knowledge. What would prevent JavaScript code from breaking any kind of static knowledge we have in the IR? The answer lies in the strict separation between what JavaScript can and cannot "see".

Taking two extreme examples, JavaScript code can obviously see the top-level exports of IR classes, but it cannot see a local variable in a method of the IR. From the specification of the run-time semantics of the IR, we can derive an exhaustive list of what JavaScript can see:

- Since JavaScript can access its global scope and perform imports, it can see the top-level exports of the IR (which are stored in the global scope or exported from a JavaScript module, according to Sections 8.1.3 and 8.1.6 of the run-time semantics, respectively):
  - The class value of a JavaScript class with `export top class`
  - The singleton instance of a Scala module class or JavaScript module class with `export top module`
  - The function value for an `export top static def` (which includes the ability to call it)
  - The ability to read a static field exported with `export top static field`

- If JavaScript holds an instance of a Scala class (including the singleton instance of a Scala module class), it can call any of the essential internal methods of that object (defined

in Section 4.1), through the respective ECMAScript Language constructs and built-in functions. Indirectly, the internal methods [[Get]] and [[Set] give JavaScript the ability to:

- Call exported member methods of the object's class, with that object as receiver

- Call the getters and/or setters of exported member properties of the object's class, with that object as receiver

- If JavaScript holds a `Char` or `Long` exotic object, it can obtain a string representation thereof, according to Section 4.1.8

- If JavaScript holds the class value of a JavaScript class, or an instance thereof, it has unrestricted access to all its exported instance and static members, as well as to its constructor, given the way such class values are created in Section 8.1.28

- If JavaScript holds a reference to a function value created in Section 9.55.7, it can call it and otherwise manipulate it as any other function value

Other than the above exceptions, JavaScript cannot see or touch the abstractions of the IR. Here are a few noteworthy examples, far from being exhaustive, of things it cannot do:

- JavaScript cannot see fields nor non-exported methods of Scala classes, even when it receives an instance of one.

- JavaScript code cannot alter the prototype chains or otherwise the structure of Scala classes.

- JavaScript cannot directly instantiate any Scala class, since that would require access to a constructor, and constructors are never exported.

- For the same reason, JavaScript cannot extend any Scala class.

Even though JavaScript has no static type system, the small set of explicitly visible things above constitutes the only entry points to an IR program, besides its *main expression*. Similarly to how the Separate Compilation Assumption [2, 1] allows to build sound call graphs for applications only (without looking at the details of libraries), the above restrictions allow to build sound call graphs for the Scala.js IR program only, in a codebase that also includes unknown JavaScript code. Further, the total absence of run-time reflection in the Scala.js IR means that analyses do not need configuration to "whitelist" some parts that would be accessed via reflection.

Consequently, the IR can be reasoned about as if living in a closed world. This property is instrumental for the optimizations that we perform in Chapter 4 to be sound.

3.4   Conclusion

In this chapter, we have precisely defined the Scala.js IR: its typing rules and its run-time semantics. We have also illustrated how it plays a role in the cross-platform language: it features both statically typed Scala classes and operations for portability, and dynamically typed JavaScript classes and operations for interoperability. All of these appear in a unified framework, without resorting to external "FFI" that break through the language semantics instead of integrating into them.

In the next chapter, we will dive into the implementation aspects for performance. This will further highlight why this tight integration of Scala and JavaScript concepts at the IR level, in an (expected-to-be) sound type system, allows for easily optimizing without fear of breaking interoperability.

Chapter 4

From the IR to JavaScript: Claiming Performance

In the previous chapter, we have given a precise definition of the Scala.js Intermediate Representation, both its type system and its run-time semantics. In this chapter, we will explore how we compile this IR into JavaScript source code that is both a) correct, in the sense that executing the JS code is equivalent to interpreting the IR and b) efficient.

We will first give an overview of the compilation pipeline in Section 4.1. In Section 4.2, we will show what kind of optimizations we can perform on the IR, including a detailed analysis on how we can perform whole-program optimizations in an incremental and parallel way. Third, in Section 4.3, we will cover the encoding of the IR into JavaScript, which is ultimately what preserves its run-time semantics when compiling to JavaScript code. We elaborate on the specific encoding of `Long`s, i.e., 64-bit integers, in Section 4.4. Section 4.5 concludes the chapter with a performance analysis of programs emitted by the Scala.js compiler.

4.1 Compilation Pipeline

With a bird's eye view, the compilation pipeline of Scala.js is straightforward:

1. Compile `.scala` files into the IR, which we store in `.sjsir` files. This step supports separate compilation (and libraries are typically published in their binary form as `.sjsir` files).

2. At link time, all the `.sjsir` files on the classpath are linked together and typechecked.

3. The optimizer performs transformations on the linked IR (optional).

4. The optimized IR is compiled into a single `.js` file by the emitter.

Because of the relatively loose definition of Scala as source language itself, let alone Scala.js, the first step cannot be specified very precisely—just like the compilation of `.scala` files to JVM `.class` files is hard to specify. However, once we have a classpath of IR files, we can build on the precise semantics from Chapter 3 to reason about optimizations and emission to JavaScript.

4.1.1 Compiling `.scala` files to `.sjsir` files

In order to compile `.scala` source files to the Scala.js IR, we use the `scalac` compiler, together with the Scala.js compiler plugin. Parsing, typechecking, erasure, and other tree transformations are delegated to `scalac`, which is an important aspect of providing portability. A few phases are sprinkled here and there in the compiler pipeline to take care of language features related to interoperability. Those phases do not touch code that exclusively manipulates Scala types, unless they define exports.

At the end of the compiler pipeline, a phase transforms the `scalac` trees into IR class definitions, which are then serialized into `.sjsir` files. This step is crucial both for portability and interoperability, as it gives meaning to the `scalac` trees. To achieve portability, this phase mirrors the corresponding phase in Scala/JVM, which generates `.class` files, using the control structures and statically typed operations of the IR. Interoperability features are compiled down to the dynamically typed operations of the IR, which can talk to JavaScript code.

4.1.2 Base Linking

Base Linking is the first step of the linker. It gathers all the `.sjsir` files on the classpath, as well as a list of so-called *module initializers* (basically `main` methods to be called, which are assembled into the *main expression* of the IR program), and establishes links between classes. As the name implies, this step is based on a reachability analysis algorithm, which identifies which classes and which of their members are necessary. Everything else is thrown away as what is usually known as interprocedural dead code elimination. We can soundly do this because of the Closed World property of the IR, which we covered in Section 3.3.2.

The result is then typechecked according to the typing rules of Appendix A. In theory, typechecking should be performed on the whole classpath, instead of the portion covered by the reachability analysis. However, doing so has problematic consequences when some classes that are part of the classpath, but not reachable, reference classes and methods that are not available. Typechecking such scenarios before reachability would fail, but succeed after reachability analysis. Now, this is a common scenario because some methods of the Scala standard library refer to parts of the JDK that are not provided by the core distribution. It is possible to use the standard library nevertheless, as long as those methods are not reachable. If the user wants to use those methods, they must bring in third-party libraries that provide the

additional parts of the JDK.

Performing reachability before typechecking also means that the reachability analysis cannot assume that the IR is well-typed. For the kind of basic reachability analysis that we do, this turns out not to be a big difficulty, besides validating that the class hierarchy is well-formed (e.g., that a Scala class does not extend a JavaScript class), independently of the members.

Once a classpath and the module initializers have been linked and typechecked, we obtain a valid Scala.js IR program, internally called a *linking unit*, which can be further processed.

4.1.3   Optimizations on the IR

This optional step is a function from linking unit to linking unit, which (hopefully) optimizes its input according to some metric. The run-time semantics of the linking unit must be preserved by the optimizer. Once again, the Closed World property of the IR, along with type soundness, is crucial for the optimizer to be able to soundly perform any non-trivial optimization, such as inlining or scalar replacement.

The optimizer implemented in the standard distribution of Scala.js is incremental and parallel. We describe it extensively in Section 4.2.

4.1.4   Emitting JavaScript code

After having optionally optimizated the IR, we can compile it down to a JavaScript source file, either a script or a module, and either using ECMAScript 2015 constructs or sticking to ECMAScript 5.1 (as requested by the user). This step must of course properly encode the specified run-time semantics of the IR in terms of the ECMAScript language. We elaborate on this step in Section 4.3.

4.2   Parallel Incremental Whole-Program Optimizer

Thanks to the Closed World property of the IR (see Section 3.3.2), we can write optimizations that leverage knowledge from the entire program, aka whole-program optimizations. Moreover, since (we conjecture that) we have a sound type system (see Section 3.3.1), we can rely on the static types of expressions in the IR for optimizations. For example, knowing the static type of a receiver, we can look up the possible targets of a virtual method call, and if there is only one possible target, inline it. Such optimizations can be very powerful, and can give Scala.js programs leverage in terms of performance and code size over more dynamically typed languages (e.g., JavaScript itself), or unsoundly typed (e.g., TypeScript), or even languages which do not assume a closed world (e.g., BuckleScript).

Since interoperability features in the IR are part of the specified semantics, the optimizer can reason about those, and still optimize methods that contain them. This would not be possible if interoperability required (or allowed) the user to embed "raw" JavaScript source code in the IR (or in the source language), as is the case in many languages compiling to JavaScript. Indeed, it would be impossible for an optimizer to correctly reason about the interactions between unanalyzed JavaScript code and the surrounding IR, forcing it to make very conservative assumptions and hence preventing some optimizations. Although most dynamically typed operations in our IR have great power (they typically can perform arbitrary global side effects), there are certain things that they cannot do, such as messing with the lexical scope of the IR, in particular the value of local `vals` and `vars`, or with the soundness of the type system.

Other than this property, the optimizations that we perform are not fundamentally novel, which is why we will not describe them further. They include standard optimizations such as constant-folding, inlining, scalar replacement, etc.

However, we will cover in significant details the incremental and parallel nature of the optimizer, which makes for the remainder of this section. This work has been done in collaboration with Tobias Schlatter, and has been published in the proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications [15]. The paper used examples based on pseudo Java source code for a broader audience. We have adapted it back to the Scala.js IR in this dissertation.

4.2.1 Motivation

Whole-program optimizations are powerful, and are used in most compilers nowadays, ahead-of-time and just-in-time alike. However, as their name implies, they require knowledge about the entire program. In other words, they are not modular, since changing one part of the program might require reoptimizing other, seemingly unrelated parts. For example, consider these two simple IR classes:

```
class A extends Object {
  val x: Int
  def init___I(x: Int) = {
    this.Object::init___();
    this.x = x
  }
  def getX__I(): int = {
    this.x
  }
  def print__V() {
    mod:Console$.println__O__V(this.getX__I())
  }
}
```

```
class B extends Object {
  def foo__A__V(a: A) {
    a.print__V()
  }
}
```

While optimizing `B.foo`,[1] we might decide to inline the body of `A.print()`, because it is so short. To do this, though, we need two pieces of global knowledge:

- The fact that no subclass of `A` overrides `print()`, otherwise inlining would break dynamic dispatch.

- The actual body of `A.print()`.

If `A` and `B` are stored in separate compilation units (files), it might be that `A` is recompiled, but `B` is not. If the body of `A.print()` changes from one compilation to the next, the optimized version of `B.foo` is outdated. Another, more subtle scenario is the *addition* of a new class `C` extending `A` and overriding `print`. This would also invalidate the optimized version of `B`. In general, other kinds of knowledge may be needed, and all kinds of scenarios might invalidate many sorts of optimizations. That is why whole-program optimizers typically work in batch mode: they start the optimization of an entire program from scratch on every compilation run, even if only a small part is changed.

This is, however, wasteful. An optimizer should ideally only reoptimize methods impacted by the change, instead of the entire program. For other methods, it could reuse the optimized version from a previous run. This is similar to compilers with separate compilation: they reuse compiled versions of source files that have not changed to speed up every compilation run. Analogously, such an incremental behavior could dramatically reduce the time spent on optimizing the program during the development cycle.

But why is it important to have a fast optimizer? Of course we want a fast resulting program for production, so we want to run whole-program optimizations at that time. Surely we do not need to do so every time we save-compile-test? We can be content with the compiled, non-optimized program for iterative development, can't we?

Indeed, in most setups, this is enough, and we believe that is why incremental whole-program optimizations have been but scarcely addressed so far. However, in some cases, this is not acceptable. For example, the compilation of Scala to Android *requires* whole-program optimizations to be performed on every run, because of limitations imposed by the Dalvik VM.

Scala.js, for its part, is a cross-breed of two languages whose development cycles are improbably dissimilar. On the one hand, Scala, with its slow compiler, its huge standard library with many indirections and higher-order methods, and a virtual machine with slow startup times, yields a relatively slow cycle where we expect the compiler (or the IDE) to detect as many

---

[1] We omit the full signatures in method names when there is no ambiguity.

errors as possible. On the other hand, JavaScript has libraries that are as small as possible, and a community expecting instantaneous save-refresh cycles. The problem is, with a non-optimized huge standard library compiled to JavaScript, the JavaScript virtual machines take noticeable time just compiling it every time the program is run. If optimizations are fast enough and bring enough speed improvements to the interpreter startup time, it is worth doing them on every development cycle. This is precisely the case in Scala.js, with incremental whole-program optimizations.

The approach can also be beneficial to other environments. In general, we believe that any rich language with big libraries compiling to a constrained environment can benefit from incremental whole-program optimizations.

Writing incremental optimizers is not a new idea. For example, Chambers et al. [8] developed a very general framework which provides fully automatic incremental reoptimizations of the program. While their ultimate goal was to reduce the time spent when reoptimizing the whole program, they do not measure nor discuss the actual running time of their compiler. What they measure instead is the selectivity of their incremental framework, i.e., how *accurate* it is, and they show that it reoptimizes very few methods. We think that this is not the most appropriate metric: even a highly accurate incremental framework is useless if deciding what to reoptimize takes longer than reoptimizing the whole program. Consider the extreme case, where the incremental optimizer first optimizes the whole program to determine what has changed with respect to the previous run. Such an incremental optimizer would be optimally accurate, yet slower than a batch optimizer. This absurd situation is of course not what we have in the case of Chambers et al.'s framework. It was implemented in the Vortex optimizing compiler and applied during the development cycle of Cecil programs, including Vortex itself. As mentioned by Dean in [13], the framework was selective enough to be used for day to day development, suggesting that it was indeed faster than a batch optimizer. However, we believe that, in aiming for the wrong evaluation metric, they might have missed opportunities on making their framework even better.

In contrast to those previous efforts, we directly focus on evaluating and reducing the running time of the entire pipeline: the detection part plus the reoptimization part. This overall running time is ultimately what the developers care about. To achieve this, we sacrifice the completely automatic nature of the incremental analysis and instead developed a methodology for designing *modular* incremental optimizers. They are modular in the sense that the change detection algorithm is isolated from the optimizer's heuristics and mechanisms and vice versa. It is therefore easy to reason about the correctness of the incremental analysis on the one hand, and the optimizer's logic on the other hand.

The remainder of this section is structured as follows. We first introduce our approach to designing modular incremental whole-program optimizers for object-oriented and functional languages. Such optimizers detect what methods of the program need to be reoptimized when some parts of the program change. The approach is based on knowledge queries,

which we introduce in Section 4.2.2. They automatically create a modular interface between changes in the program and the optimizations they invalidate. We show how the approach accommodates a variety of type-based whole-program optimizations for object-oriented and functional languages in Section 4.2.3: inlining with static and dynamic dispatch, elimination of subtyping checks, scalar replacement, and closure elimination. We evaluated our approach in the context of the incremental whole-program optimizer for Scala.js. We show how the general algorithm can be made parallel in Section 4.2.5. Results presented in Section 4.2.6 show that incremental runs are 10 to 100 times faster than batch runs, and that the parallel algorithm scales with the number of threads.

4.2.2 Knowledge Queries

As hinted in the previous section, the difficult part of an incremental optimizer is to detect which methods need to be reoptimized when certain parts of the compiled (non-optimized) program change, i.e., tracking dependencies between optimized methods and the program.

We introduce our approach by studying the simple case of inlining static methods. Static methods are always referred to with their full name, and do not need polymorphic dispatch resolution. In Section 4.2.3, we will progressively lift those restrictions, and show how other kinds of optimizations map into our approach.

A Pure and Restricted API

Our solution is based on one key idea: provide a restricted API through which the optimizer can query facts about the whole program. We call the functions in this API *knowledge queries*. They must take immutable arguments and return immutable results, which may depend only on the arguments and the program. Knowledge queries are therefore functions of their arguments and the program.

Our optimizer works on a per-method basis. An instance of the optimizer is created for every method, and is tasked to work solely on this method. Hence, we name it a method optimizer. To gather any information about the program other than the method itself, a method optimizer can only use knowledge queries. Its result (the optimized method) must therefore be a function of two things:

- the non-optimized method, and
- the results of knowledge queries.[2]

For example, consider a variant of the program from the introduction where everything is static, and with an additional method `C::bar__V()`.

---

[2]Pragmatically, it could depend on other factors such as randomness or time spent. The point is that it cannot depend on the program other than through knowledge queries.

```
class A extends Object {
  static var x: int
  static def getX__I(): int = {
    A::x
  }
  static def print__V() {
    mod:Console$.println__O__V(A::getX__I())
  }
}
class B extends Object {
  static def foo__V() {
    A::print__V()
  }
}
class C extends Object {
  static void bar__V() {
    mod:Console$.println__O__V("bar: " +[string] A::getX__I());
  }
}
```

The method optimizer for `B::foo` initially knows nothing about the program, except the body of that method. As it processes the body, it finds the call `A::print__V()`, which it may decide to inline. Typically, inlining decisions are based on properties of the *target* method as much as the caller. The optimizer already knows about the caller, but not the target.

Since static methods are linked statically, we know that the target method must be the method `print` in the class `A`. But that is all the optimizer can know about the target (since it does not know the program) without calling a knowledge query. Suppose we base our inlining decisions on the body of the target method. In this case, we need the following query:

```
def getMethodBody(method: MethodID): Tree
```

where `MethodID` is the fully qualified name of a method in the program (here, `"A::print__V"`). Once the optimizer knows the body of `A::print__V()`, it can decide whether to inline it or not (e.g., based on its size). Smarter heuristics can be used, such as those developed by Sewe et al. [57]. If (and only if) it decides to inline `A::print__V`, it will process that method's body for optimizations too. While doing so, it will encounter the call `A::getX__I()`, which it will also consider for inlining. It will therefore also invoke the query `getMethodBody("A::getX__I")`. It will not do so if it decides not to inline `A::print__V`. The set of knowledge queries performed by an optimizer may therefore depend on its optimization decisions: it is not an inherent property of the program.

An Automatic Dependency Tracker

Since the result of optimizing a method m must be a function of its non-optimized body and the knowledge queries, the latter are a natural and automatic dependency tracker, which we can use for incremental reoptimization. If the non-optimized version of m and the results of

```
getMethodBody("A::print__V") ●          ● A::print__V
  getMethodBody("A::getX__I") ●          ● A::getX__I
  getMethodBody("B::foo__V") ●          ● B::foo__V
  getMethodBody("C::bar__V") ●          ● C::bar__V
```
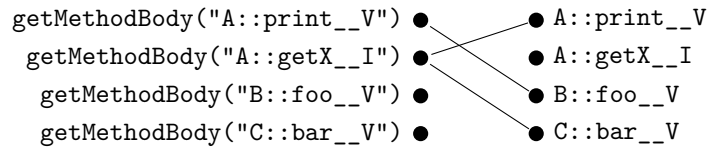
Figure 4.1 – Dependency graph if `A::print__V` is not inlined

```
getMethodBody("A::print__V") ●          ● A::print__V
  getMethodBody("A::getX__I") ●          ● A::getX__I
  getMethodBody("B::foo__V") ●          ● B::foo__V
  getMethodBody("C::bar__V") ●          ● C::bar__V
```
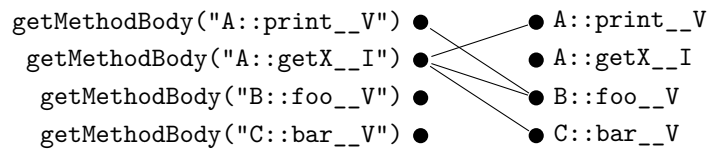
Figure 4.2 – Dependency graph if `A::print__V` is inlined

all knowledge queries performed by its optimizer are the same for two successive versions of the program, then the result of the optimizer must be the same too. Hence, we need not reoptimize *m* for the second run of the incremental optimizer: we may reuse the optimized method from the previous run.[3]

This forms the basis for the incremental optimizer based on knowledge queries:

- While optimizing a method *m*, we log all the knowledge queries invoked by the optimizer, and register them as dependencies for the optimized *m*.

- On the next run of the incremental optimizer, we detect which knowledge queries have different results than in the previous run, and we invalidate all the optimized methods that depend on them.

The recorded dependencies form a bipartite graph with two kinds of nodes: knowledge queries (including their arguments), and optimized methods.

Figure 4.1 shows the dependency graph that we get for the program of the previous section if the optimizer for `B::foo__V` decides not to inline `A::print__V`. If it does inline `A::print__V`, then we get the graph of Figure 4.2, in which there is an additional edge between `B::foo__V` and `getMethodBody("A::getX__I")`.

Program Changes

We still miss one piece of the algorithm: how do we detect which knowledge queries have different results? For this, we compute program changes, and their impact on knowledge queries.

A *program change* is a difference between the program in the current run compared to the

---

[3]If we accept randomness and other independent factors, we must reformulate this as: the result for the previous run is *a* possible result for the second run as well, which is also sufficient to allow us to reuse it.

previous run. Program changes can be of any granularity, but in practice we only consider class-level and method-level changes. Tracking dependencies at the instruction level is useless for one simple reason: as programs get bigger, they contain arbitrarily more classes and more methods, but the size of each method tends to be bounded, due to established programming best practices. This means that we need not scale with the size of methods, but rather with the number of classes and methods in the program. Section 4.2.4 shows how we compute the changes we are interested in.

Intuitively, a program change *impacts* a knowledge query if it changes the result of that query. Hence, for each change, we have to compute which queries it impacts. All the methods depending on those queries (which we get from the dependency graph) must be invalidated.

A little bit more formally, we say that a program change impacts a knowledge query if and only if there exists a program P such that the result of that query is different for P than on the program P' obtained by applying the change to P. By this definition, and because of the commutativity of difference, a program change always impacts the same set of queries as its reciprocal.

The changes we need to detect are derived from the queries used by the method optimizer. So far, we have only one query: `getMethodBody`. The only program change we need to cover is straightforward:

- Change the body of a method `C.m` (or `C::m` for static methods).

which always impacts only `getMethodBody(C.m)`. We will see in Section 4.2.3 that not all knowledge queries translate that straightforwardly to program changes.

Unlike the dependency tracking between the optimizer and the knowledge queries, the relation between program changes and their impacts is not automatic: it must be reasoned about manually. However, the big advantage is that the details of the optimizer, which are bound to be complex, are completely abstracted away from that reasoning by the knowledge queries, which are automatic. The knowledge queries therefore create a truly *modular* interface between the optimization logic and the incremental logic.

We conclude this section with the global view of the whole incremental algorithm. It has two phases per run:

- Invalidation phase: diff the program compared to the previous run to isolate *program changes* (see Section 4.2.4). For each program change, compute the set of *impacted* knowledge queries (see Section 4.2.3). Follow the edges in the dependency graph to invalidate all the methods depending on these queries.

- Optimization phase: run the method optimizer for every invalidated method. First remove all edges in the graph for this method. Then, optimize its body and record all invoked knowledge queries as edges in the graph.

4.2.3 Knowledge Queries for OO and Functional Languages

In the previous section, we introduced knowledge queries with the case of inlining static methods only. We now add language features and optimizations with two goals in mind:

- show how knowledge queries guide the design of incremental optimizations, and

- show how a variety of whole-program optimizations for object-oriented and functional languages fit in the knowledge query approach.

First, we add language features, while still studying their impact on inlining only, until we can handle all the Scala features of the Scala.js IR object model: single inheritance, interfaces, dynamic dispatch and run-time instance tests. Then, we introduce support for other kinds of whole-program optimizations: elimination of run-time type tests, scalar replacement and closure elimination.

Inheritance

In Section 4.2.2, we have silently omitted to consider *inherited* static methods. In Java, as well as most object-oriented languages, static methods defined in a superclass `A` can be called on a class `B` when `B` extends `A`. In the Scala.js IR, that is not allowed, but there is another feature with similar characteristics: the `ApplyStatically` nodes of the form `obj.C::m()`, which statically dispatch to an instance method of an explicitly given class. When calling `obj.B::foo`, the actual target is looked up in the parent chain of `B` on a first match basis. Overridden methods therefore shadow methods coming from the superclasses.

For example, consider the following snippet:

```
class A extends Object {
  def foo__I(): int = {
    3
  }
}

class B extends A {
}

class C extends B {
  def foo__I(): int = {
    5
  }
}

def bar__C__V(c: C) {
  println(c.A::foo__I());
  println(c.B::foo__I());
  println(c.C::foo__I());
}
```

in which the method `bar__C__V` calls the method `A.foo__I` twice then the method `C.foo__I` once. The call `c.B::foo__I()` resolves to `A.foo__I`, and it is therefore not possible any more to identify the actual target only looking at the call site. Some global knowledge is required.

There are other cases where static call resolution applies:

- Calls of static methods in Java, because they are inherited in subclasses, and can be shadowed by redefinitions.

- Non-virtual methods in C++.

- `super` calls in Java and Scala (which cause `ApplyStatically` nodes to be created in the Scala.js IR).

- In an optimizer, we can use static resolution even if the language prescribes dynamic resolution, provided we know the exact class of the receiver (e.g., because it is allocated with `new` within the scope of the optimizer).

We will now extend the support for incremental inlining from the previous section to handle such resolutions.

Eliciting Knowledge Queries   The first step is to determine what knowledge queries are necessary. To do this, we simply update the method optimizer. When it needs some information about the whole program, we introduce a new knowledge query.

Here, we want to inline calls such as the following:
```
obj.T::m()
```

Because of inheritance, `m` could be declared in a a superclass of `T`. Finding the exact (unique) target depends on whole-program knowledge. We therefore introduce a new knowledge query:
```
def resolveStaticCall(classID: ClassID, methodName: String): MethodID
```

The optimizer can use this knowledge query to retrieve the actual target of a static call. This query will typically be followed by the existing query `getMethodBody(target)` to decide whether to inline that target, and if yes, how.

Impact of Program Changes on Queries   We now need to efficiently find queries that are impacted by program changes. Let us start with an example. In one run of the optimizer, the program looks like this:
```
class A extends Object {
  def foo__V() {
    mod:Console$.println__O__V("A")
  }
}
class B extends A {}
class C extends B {}
```

```
class Main extends Object {
  def main__V() {
    val obj: C = new C
    obj.C::foo()
  }
}
```

While optimizing `Main.main`, we consider the call `C.foo()` for inlining. The semantics of static calls tell us that the actual target method is `A.foo`. Therefore, `resolveStaticCall(C, "foo")` returns `"A.foo"`.

Now, in the next run, we *add* the method `B.foo`:

```
class B extends A {
  def foo__V() {
    mod:Console$.println__O__V("B")
  }
}
```

Assuming we have a perfectly accurate incremental *compiler*, this change does not impact recompilation of `Main`, since the public API of `C` has not changed.[4] However, it must prompt reoptimization of `Main.main`, since the actual target method changes: now it is `B.foo`. In other words, the result of the knowledge query `resolveStaticCall(C, "foo")` has changed from `"A.foo"` to `"B.foo"`. Hence, the addition of a new method `B.foo` impacts this knowledge query. In general, adding a method `X.m` will impact `resolveStaticCall(Y, "m")` for all classes `Y` extending `X`, directly or indirectly. Since no other method in the program asked for `resolveStaticCall(C, "foo")`, the addition of `"B.foo"` does not trigger reoptimization of any other part of the program.

Removing a method will have the same impact, since it is the reciprocal of adding it.

In a program with inheritance, we identify the following possible core changes:

- Change an existing method's body

- Add/remove a method in an existing class

- Add/remove an empty class without child classes (another pair of reciprocal changes)

Other changes can be represented as composition of these changes:

- Adding a non-empty class is equivalent to adding it empty, then adding methods.

- Similarly, removing a non-empty class is equivalent to removing its methods, then removing it.

- Changing a class' parent, i.e., moving it in the hierarchy, is equivalent to deleting the class and all its subclasses from the previous parent, and adding them back to the new

---

[4]This depends on the format and specifications of the compiled files. For example, it is not true for `.o` files in C++. For the `invokespecial` instruction of the JVM, it is true.

parent.

Further, note that when a class is removed (including when it is moved), all its subclasses are removed as well.

We have already seen the impact of the two first core changes. Adding an empty class `C` that does not have any child class does not impact either `getMethodBody` (since it does not have any method) nor `resolveStaticCall` (because no existing code could possibly have a call `C.m()`, since `C` did not exist in the previous run). It follows that removing an empty class, which is the reverse operation, does not have any impact either.

As a recap, we have the following:

- Changing the body of `C.m` impacts `getMethodBody("C.m")`.

- Adding or removing a method `C.m` impacts `resolveStaticCall(D, "m")` for all classes `D` inheriting from `C`.

That is it. With a simple two-step design, we added support for inheritance. First, we listed knowledge queries needed by the optimizer. Then, we identified which program changes can impact these queries.

Polymorphic Dispatch

So far, we have limited our discussion to statically dispatched calls. However, the IR, just like Java, Scala, and other object-oriented languages, has virtual calls, where dynamic dispatch is required. The exact target of a call is not known at compile-time, since it depends on the run-time type of the receiver. To support inlining with polymorphic dispatch, we again find out what knowledge queries are needed, and what program changes impact them.

Eliciting Knowledge Queries   When encountering a dynamic call of the form `x.m()`, the optimizer has to determine the target of that call. Unlike with static calls, there can be multiple targets, if `x` is of a type `C` that has several subclasses overriding `m`. We therefore need the following new knowledge query:

```
def resolveDynamicCall(classID: ClassID,
    methodName: String): Set[MethodID]
```

With the set of possible targets, the optimizer can decide whether or not to inline (e.g., if the set is a singleton). As was the case with static calls, it can follow up with `getMethodBody` to obtain the body of a given target to inline.

Impact of Program Changes on Queries   Consider the following base program with an instance method `foo`:

```
class A extends Object {
  def foo__V() {
    mod:Console$.println__O__V("A")
  }
}
class B extends A {}
class C extends B {}
```

and consider the calls `a.foo()` and `c.foo()`, with a (resp. c) statically typed as A (resp. C). The corresponding knowledge queries, `resolveDynamicCall(A, foo)` and `resolveDynamicCall(C, foo)`, both return the singleton {"A.foo"}.

We now add a method `B.foo`. Similarly to the query `resolveStaticCall`, `resolveDynamicCall(C, foo)` is impacted, since its result becomes {"B.foo"}. However, in this case the query `resolveDynamicCall(A, foo)` is also impacted, with a result of {"A.foo", "B.foo"}. This follows from the fact that B <: A. So a value statically typed as A can hold an instance of class B.

In general, adding or removing a method `X.m` impacts the queries `resolveDynamicCall(Y, m)` for all Y such that either Y <: X or X <: Y.

Interfaces

The addition of interfaces (without default methods) complicates dynamic calls. Now, in a dynamic call `x.m()`, x can be statically typed as an interface I. Consider the following program:

```
interface I {
  def foo__V()
}
class A extends Object {
  def foo__V() {
    mod:Console$.println("A")
  }
}
class B extends A {}
class C extends B implements I {}
```

and the call `x.foo()` where x is statically typed as an I. The result of the knowledge query `resolveDynamicCall(I, "foo")` would return the singleton {"A.foo"}.

Let us now add a method `B.foo`. This changes the result of the query from {"A.foo"} to {"B.foo"}, even though neither B <: I nor I <: B. Here, the query is impacted because there exists a subclass C of B that implements I.

In general, adding (or removing) a method `X.m` now also impacts the knowledge query `resolveDynamicCall(I, m)` for all I such that there exists a class Z such that Z <: X and Z <: I. Taking a step back, the rule for classes that we saw in the previous section is a special

case of this one. We therefore combine both rules as one: adding (or removing) a method `X.m` impacts `resolveDynamicCall(Y, m)` for all classes and interfaces `Y` such that there exists a class `Z` such that `Z <: X` and `Z <: Y`. This looks heavy to compute, but we can easily reformulate it as follows: for all subclasses `Z` of `X`, for all ancestors `Y` of `Z`, `Y.m` is impacted. If we maintain a data structure that allows us fast access to all subclasses of a class, and all ancestors of a class, the computation becomes straightforward.

We are not done with interfaces, though. The addition of interfaces to the language also introduces a new kind of program change:

- Add/remove an interface `I` to the ancestors of a class `C` (pair of reciprocal changes).

Adding `I` to the ancestors of `C` implies that, now, `C <: I`. A variable `x` of type `I` can therefore hold a value of class `C`, where previously it could not. In terms of dynamic calls, this means that the target of `x.m()` can change for any `m`. This change (and its reciprocal) will thus impact the queries `resolveDynamicCall(I, m)` for all `m`.

Note that it is not necessary to track the methods defined in the interface itself with this approach.

Other Object-Oriented Features

There are a couple of other typical object-oriented features that do not require any additional support.

Multiple inheritance is entirely covered by the above treatment of interfaces, as far as knowledge queries are concerned.

Overloading is a compilation issue. When they reach the optimizer, overloaded methods have already been disambiguated, either with mangled names (e.g., in C++ or the Scala.js IR) or because they are identified by their full signature (such as on the JVM).

Similarly, operator overloading is transformed by the compiler into method calls, and is therefore covered.

Eliminate Subtyping Checks

Now that we have a full object model of the IR, we can move on to support other kinds of optimizations, besides inlining. We begin with eliminating runtime subtyping checks, i.e., `isInstanceOf` and `asInstanceOf`. Due to other whole-program optimizations, such as inlining, we can be left with tautological subtyping checks, such as

```
interface Foo {}
class Bar extends Object implements Foo {}
```

```
val x: Bar = ...
x.asInstanceOf[Foo]
```

Because `Bar <: Foo`, the cast is redundant, and can be eliminated, giving `x`. Similarly, a test `x.isInstanceOf[Foo]` can be optimized as `x !== null`. However, to do this, we need to know that `Bar <: Foo`, which is global knowledge of the program. It must therefore be requested as a knowledge query.

Eliciting Knowledge Queries   When encountering `x.isInstanceOf[Foo]`, with x of type `Bar`, the optimizer must test whether `Bar <: Foo`. The obvious knowledge query is therefore the following:

```
def isSubclass(subclass: ClassID, superclass: ClassID): Boolean
```

with the understanding that a `ClassID` can also refer to an interface.

Impact of Program Changes on Queries   The query `isSubclass(subclass, superclass)` only depends on the list of all ancestors (classes and interfaces) of `subclass`. More specifically, whether `superclass` is in this list or not. Because the parent chain of a class cannot change (it would be removed and added instead), only interfaces can be added to or removed from the ancestor list. We extend the program change we introduced with interfaces:

- Add/remove an interface `I` to the ancestors of a class *or interface* `J` (pair of reciprocal changes).

which now also impacts the query `isSubclass(J, I)`.

Scalar Replacement

Scalar replacement, also known as stack allocation (although they are not exactly the same thing), is an optimization that replaces a reference value (to an object allocated on the heap) by a set of values for all fields of the given object. For example, replacing:

```
class Point extends Object {
  val x: double
  val y: double
  def init___D__D(x: double, y: double) {
    this.Object::init___();
    this.x = x;
    this.y = y
  }
  def abs__D(): double = {
    mod:Math$.sqrt__D__D(
      this.x *[double] this.x +[double] this.y *[double] this.y)
  }
}
```

```
def foo__V(y: double) {
  val point: Point = new Point.init___D__D(5, y);
  mod:Console$.println__O__V(point.x);
  mod:Console$.println__O__V(point.abs__D())
}
```

by

```
def foo__V(y: double) {
  val point_x: double = 5;
  val point_y: double = y;
  mod:Console$.println__O__V(point_x);
  mod:Console$.println__O__V(mod:Math$.sqrt__D__D(
    point_x *[double] point_x +[double] point_y *[double] point_y))
}
```

Note that we inlined the call to `point.abs()`. If we cannot inline `point.abs()`, the optimization is canceled, since the `Point` must be allocated to call `abs`. Instead, we could use partial escape analysis as developed by Stadler et al. [60].

This optimization improves several aspects:

- Memory consumption and GC pressure, because less objects are allocated

- Execution speed, because less pointer indirections are involved

- It is an enabler for other optimizations, because we can often have more precise static information on the fields (e.g., here, we can constant-fold `point_x`).

Scalar replacement obviously needs global knowledge: what fields are defined in the class `Point`, as well as the body of its constructor. We can use knowledge queries to introduce this optimization in an incremental framework.

Eliciting Knowledge Queries Although scalar replacement has several implications down the line, its need for global knowledge is actually confined to the `new` invocation. Indeed, once the allocation is replaced by individual variables, the fact that `point` has been scalar-replaced into `point_x` and `point_y` becomes part of the local state of the optimizer.

To keep things minimal, we decompose an allocation such as `new C.init(args)` into the allocation of the object of class `C` itself, and the call to its constructor `init`. The latter is a standard application of a static call, as described in Section 4.2.3. The interesting part is the allocation. To replace it, we only need to know its fields. The body of the constructor itself is not needed, as it is part of the static call. We therefore derive the following knowledge query:

```
def getScalarReplacement(classID: ClassID): List[Field]
```

where `Field` is a description of a field, with its type and potentially other properties, such as mutability.

Impact of Program Changes on Queries   We introduce the following program change:

- Change the fields of a class `C` (including those inherited)

which impacts the query `getScalarReplacement(C)`.

Closure Elimination

Closure elimination is an important optimization for languages with higher-order functions, obviously including functional languages. We do not need any other query (and therefore no other program change) to support this optimization. Indeed, closure elimination derives from inlining, local constant propagation, and either beta-reduction or scalar replacement, depending on whether the IR supports closures directly or encodes them as anonymous classes. All of these optimizations are either local optimizations, or we have already shown how to support them. Hence, closure elimination is trivially supported by our approach.

4.2.4   Diffing the Program Between Runs

In the previous sections, we showed how knowledge queries apply to a variety of whole-program optimizations. To do so, we relied on *program changes* as small differences between two versions of a program. As a recap, here are the various kinds of changes we relied upon:

- Change the body of a method `C.m`

- Add/remove a method `C.m`

- Add/remove an empty class `C` without child classes

- Add/remove an interface `I` to the list of ancestors of a class or interface `J`

- Change the fields of a class `C`

If the earlier steps of the compiler were incremental themselves, they could communicate these changes directly to the incremental optimizer. However, this is not the case in Scala.js: the smallest unit of change that the optimizer receives is a compiled class (or interface), each being stored in a compiled `.sjsir` file. In this section, we sketch how to diff two object-oriented programs so that we can derive a list of the above changes.

To guarantee the correctness of the incremental optimizer, we need to make sure that we produce an *exhaustive* list of program changes. If we miss one, we might not detect the impact of changes on some knowledge queries, which in turn, would cause some optimized method not to be invalidated when they should, making the optimizer unsound. It is not necessary for correctness to produce a *minimal* list of program changes, i.e., we are allowed to emit more changes than necessary, although that would invalidate more methods than required. We

do not argue for the precision of our diff, and instead focus on the overall running time in Section 4.2.6.

Since every program change is related to some class or interface `C` or `J`, we can divide and conquer the problem by class/interface. For each class or interface in the program, we seek to produce an exhaustive list of program changes related to it.

To compute our structured diff, we maintain two data structures. First, a map from each class/interface `J` to the set of its ancestors. Second, a tree of the classes in the program mirroring the class hierarchy. Each class keeps track of:

- its parent and child classes, forming the tree structure
- its methods and their bodies
- its fields, including inherited ones

Together, those two data structures completely define the entire program. Computing the program changes caused by changes to these structures therefore takes all possible changes to the program into account. Other than in the ancestor map, interfaces and their member methods are not part of this program representation, because they are neither used by the optimizer nor the code generator that follows it, and are therefore irrelevant at this point.

If mixed with other changes, additions and removals of classes are hard to get right, especially because of classes that move around the hierarchy. Therefore, the diffing algorithm uses 4 main steps:

1. Class deletions: remove classes that existed in the previous run, but do not exist in the new version of the program. Recall that this includes classes that are moved in the hierarchy.
2. Changes to the sets of ancestors.
3. Class changes: add and remove methods, change method bodies, and change fields.
4. Class additions: add classes that did not exist in the previous run, but do exist in the new version of the program. Again, this includes classes that have moved.

This separation also allows for a simple and efficient implementation of the batch mode (the first run, when we come from an empty program): simply run the last step.

Note that there is no step dealing with interface addition and removal. It turns out this is not necessary, since no program change is dependent on those changes.

Steps for Class Additions and Removals.   The first step is a post-order traversal of the hierarchy tree. For each deleted class `C`, we first remove all its methods (emitting "Remove a method `C.m`" changes), then remove the class itself (emitting "Remove an empty class `C` without child classes"). Since classes moving around the hierarchy are considered deleted then added, we

know that if class C is deleted, so are all its subclasses. Therefore, in a post-order walk, it indeed has no child classes anymore. There cannot be any other program change for C, so we have exhaustively listed all program changes for C.

Similarly, the last step for class additions is a pre-order traversal of the hierarchy tree. By a similar argument, we list "Add an empty class C without child classes" and "Add a method C.m" program changes for C, and no other.

Ancestors Update Step. In this step, we update the sets of ancestors of all classes and interfaces in the program. Since classes are not removed, added or moved during this step, only interfaces can be added to or removed from the sets of ancestors. For each class or interface J in the new program, we compute its new set of ancestors by transitively following the direct parent class and/or implemented interfaces. Unless J did not exist in the previous program, we diff this set with the old set of ancestors. For each difference I, which must be an interface, we emit the program change "Add/remove an interface I to the list of ancestors of J".

Since the interfaces implemented by a class do not influence its methods nor its fields, a change of ancestors need never emit program changes other than "Add/remove an interface I". Therefore the emitted list of program changes for changing the ancestor list is exhaustive.

Class Changes Step. The third step is more complicated, and takes care of all the changes in classes that are neither removed, nor added (nor moved around the hierarchy). Obviously, in this step, no program change "Add/remove an empty class C without child classes" is emitted.

For the other program changes, we walk the class hierarchy data structure. For each class node C in the class hierarchy, we compute the values of its new constituents, and emit the appropriate program changes. In this step, the parent class and the list of child classes cannot change.

The new values of the constituents of a class C are computed from a mix of information coming from the compiled file for C, and other parts of the class hierarchy data structure. By diffing the old and new values, we know what program changes to emit. Here is what happens for each of the three constituents of C:

- The parent class and child classes are never modified, since during step 3, no class is added, removed, or moved around the hierarchy.

- The set of methods depends only on the content of the compiled file for C. Diffing the set itself emits "Add/remove a method C.m" changes, while for the bodies of changed methods, we emit "Change the body of a method C.m".

- The list of fields depends on the fields of the parent class in addition to those directly declared in C. If the list of fields changes at all, we emit "Change the fields of a class C".

Since the list of methods of C does not influence fields in C and conversely, nor the set of ancestors, there is no other program change that need emitting, therefore we have exhaustively listed the appropriate program changes. Since only those two constituents can change in step 3, the above changes entirely characterize the changes in class C, and the algorithm produces an exhaustive list of all the program changes for C. Finally, since we process all classes in this way, and since we have initially divided the set of all program changes per class C or class/interface J, we conclude that we produce an exhaustive list of all the program changes for the entire program.

Note that there are data dependencies between the processing of each class, since some parts of the algorithm read data from the rest of the class hierarchy data structure, which imposes constraints on the order in which we process classes. When processing a class node in the class hierarchy, we perform the following kinds of data manipulations:

- Read the file for that class, and that class only.

- Read data in its parent classes in the class hierarchy.

- Read and write data stored in this class.

In particular, observe that:

- we do not read data from child classes, nor any other classes not in the parent chain, and

- we modify data only in the class node being processed.[5]

Therefore, there are only top-down data dependencies, i.e., from parent classes to their children. A pre-order traversal of the class hierarchy guarantees that each class is processed after all its parent classes, and that all data dependencies are satisfied.

Adding multiple inheritance to the language (or default methods à la Java 8) turns the hierarchy tree into a directed acyclic graph. A class must therefore be processed only when *all* its parents have been processed. Adding or removing a parent class must be viewed as moving the class in the graph, and must therefore be deleted and added again. The algorithm is otherwise unaffected.

4.2.5   Parallel Implementation

To take advantage of multicore architectures, we parallelized the incremental algorithm. The two phases of the algorithm (invalidation and reoptimization) must run sequentially with respect to each other, i.e., the invalidation phase must complete before the reoptimization phase can start. However, we can parallelize within each phase. Doing so turns out to be easy.

---

[5]Recall that, during step 3, no class is added or removed, hence the list of child classes is not modified.
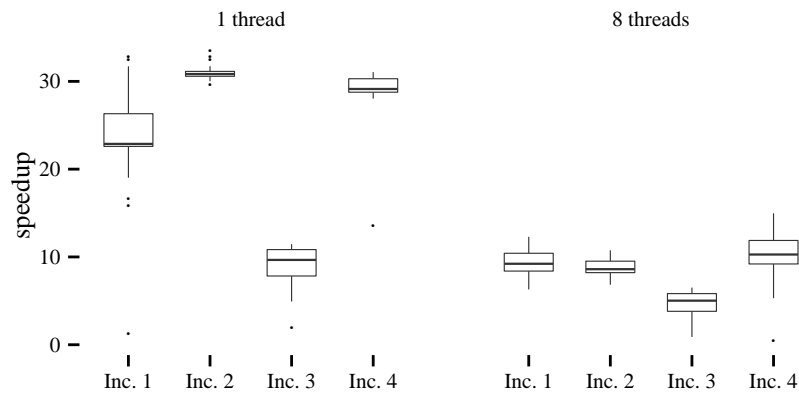
Figure 4.3 – Speedups in incremental mode with respect to batch mode

Parallelizing the Invalidation Phase

The first phase is trivial to parallelize. We already established that data dependencies flow top-down in the class hierarchy. Moreover, this phase does not modify the dependency graph. We can therefore parallelize trivially down the inheritance tree (or down the graph if we have multiple inheritance).

Parallelizing the Reoptimization Phase

The second phase is a bit less easy to parallelize, but not much. We receive a set of methods that need to be reoptimized from the first phase. Recall that every method is processed by a different optimizer instance, which holds only state local to itself. Any knowledge of the program it needs must be requested through knowledge queries. Moreover, the result of every query does not change during one run. We would like to run all optimizer instances in parallel.

Since knowledge queries read data that are immutable for the duration of the second phase, the computation of their result need not be synchronized. The dependency graph is the only shared mutable state.

Since an optimizer for a given method m only removes and adds edges (m, q), i.e., for its own method, different optimizers cannot act on the same edges. Moreover, during this phase, the graph is never read, only written to. It is therefore sufficient to implement the graph with a concurrent data structure that supports atomic addition and removal of edges, while keeping the order of operations coming from a single thread. This can easily be done with the non-blocking concurrent hash tries developed by Prokopec et al. [52].

4.2.6 Results

We evaluated the implementation of the parallel incremental algorithm in Scala.js along the following two axes:

- How the incremental version improves over the batch version of the same program.

- How the parallel implementation scales with the number of threads.

When benchmarking, we measure the running times of the invalidation phase and the re-optimization phase separately. Since the invalidation phase would not be needed at all if the algorithm did not have to support incremental compilation, we compare the time of the reoptimization phase in batch mode to the cumulative time in incremental mode. This is actually slightly biased in favor of the batch mode, since part of the job of the invalidation phase is also to construct the class hierarchy data structure, used to resolve calls, and in general answer knowledge queries.

Batch Mode versus Incremental Mode

Comparing the running time of the batch and incremental modes is the most important aspect of our contribution. We have measured this with two very different approaches: on the one hand, in a controlled environment with synthesized changes, and on the other hand, in real life by actual users of the optimizer during their day-to-day development.

Measures in a Controlled Environment   To evaluate the speedup obtained from running the incremental optimizer rather than a batch optimizer, we took the codebase of Papa Carlo [35]. This codebase contains 4974 reachable methods (the number of methods that the batch mode has to optimize), which is a relatively small though non-trivial Scala.js codebase. In batch mode, we link the full program with a clean optimizer. In incremental mode, we first optimize a slightly changed version of the program (with a fresh optimizer) and then optimize the original version of the program. This ensures that the resulting program is the same for batch and incremental mode.

We performed the measurements on an Intel Core i7-3770K clocked in at 3.5GHz, allocating 4GB of RAM to the JVM running the experiments. This CPU has 4 cores able to run 8 threads.

We compare the running time of the batch mode against the following synthesized incremental changes to the program:

1. Modify the body of an inlineable method

2. Modify the body of a non-inlineable method

3. Add a method used in a polymorphic dispatch

	# meth.	time [ms]		speedup		MAD [ms]	
Batch	4974	246	(97)	1	(1)	3.2	(27.3)
Inc. 1	2	11	(11)	23	(9)	1.7	(1.6)
Inc. 2	1	8	(11)	31	(9)	0.1	(1.2)
Inc. 3	8	25	(19)	10	(5)	5.1	(4.9)
Inc. 4	17	8	(9)	29	(10)	0.2	(1.9)

Table 4.1 – Batch mode versus incremental mode for various changes. Running with 1 thread (8 threads).

4. Mark an existing class as eligible for scalar replacement

These changes may seem arbitrary, and it might appear that using the history of a version control system (VCS) would give more realistic results. However, this is a fallacy, because commits in a VCS are much too coarse, with respect to what our incremental optimizer is trying to achieve. In a typical development workflow, we run a compile-optimize-run/test cycle many times to eventually create a single commit. It is common practice to perform continuous testing (running unit tests on every file save) of the optimized build with Scala.js, since it requires no setup.[6] The time between two cycles might go from a few seconds to a couple of minutes, while the time between two commits can cover much longer development times. VCS commits are therefore not good representatives at all. These very small synthesized changes are in fact much closer to the reality.

Figure 4.3 shows the speedups of incremental mode with respect to batch mode for the sequential version of the algorithm and the parallel version with 8 threads (absolute running times of batch mode are in Figure 4.5).

Table 4.1 shows the number of methods that were invalidated, median running time, speedup factor and median absolute deviation (MAD) for the running time.

For the single-thread case, we observe speedups up to a factor of 30 for Inc. 2 and Inc. 4. The lowest gain can be seen on Inc. 3, which only exhibits a speedup of 10x. Multi-threaded setups offer smaller speedups, because there is more contention on the parallelization of the first phase, which causes the batch optimizer to benefit more from multiple threads, relatively to the incremental optimizer. Improving the parallelism of the first phase will be future work, this mostly involves improving current work stealing techniques. It is remarkable that despite this, we can still observe speedups up to a factor 10.

Real Life Benchmarks The previous section analyzed reproducible benchmarks in a very controlled environment. However, the real purpose of our incremental optimizer is to reduce the time actual developers have to spend waiting during their development cycle. We have therefore benchmarked the optimizer in real life situations: the developers of 6 Scala.js

---

[6]The build command `sbt ~test` performs optimized continuous testing by default.

	name	# methods	# data points
A	scalajs-games demo	5,000	414
B	react-ext test suite	9,500	31
C	ScalaCSS playground	9,000	54
D	Scala.js land	5,500	42
E	Scala.js land admin	9,500	32
F	Anon. web app client	20,000	41

Table 4.2 – Codebases benchmarked in real life.
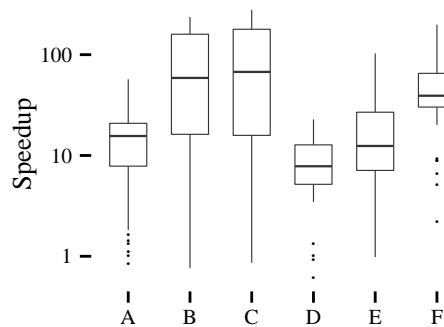
Figure 4.4 – Speedup in incremental mode wrt. batch mode (log scale, higher is better)

codebases have used an instrumented version of the optimizer for several days during their day-to-day programming tasks. The instrumented optimizer measured the running times of both the incremental and batch optimizer on the codebase every time it was invoked (up to several times per minute on some codebases).

These benchmarks were run in unknown and uncontrolled environments, during normal development tasks. We therefore expect that other processes were running at the same time. Moreover, every measure is unique and non reproducible.

Although non reproducible, we consider these measurements much more important in practice, since they are real life usages of the optimizer.

Table 4.2 lists the 6 codebases that have been benchmarked. The number of methods is a rough approximation, since it changes from one measurement to the next, and serves only as an estimate of the size of the codebase. Figure 4.4 shows, for each codebase, the speedup of the incremental optimizer wrt. the batch optimizer. Table 4.3 shows the associated numeric data.

On these benchmarks, we can observe speedups from 10x to 100x, which is a huge improvement given that these are real life measurements.

	Batch [s]	Inc [ms]	speedup
A	0.17 ± 0.0	10.5 ± 4	16 ± 9
B	3.08 ± 0.3	24.5 ± 27	59 ± 68
C	4.43 ± 0.3	65.9 ± 63	67 ± 83
D	0.28 ± 0.2	35.0 ± 19	8 ± 5
E	0.44 ± 0.4	42.9 ± 44	12 ± 10
F	3.74 ± 0.4	92.9 ± 47	39 ± 30

Table 4.3 – Measurements on real life codebases (median ± MAD)
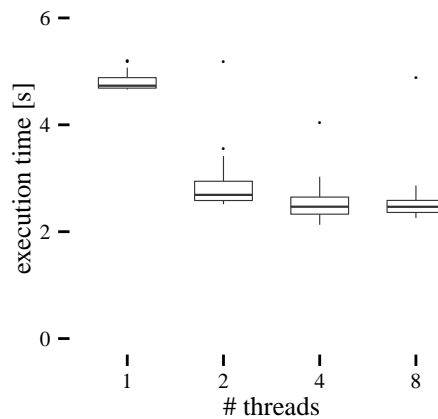
Figure 4.5 – Batch mode running times

Scaling with the Number of Threads

We also compare the batch mode running times with different numbers of threads. These measurements were performed on the same Intel Core i7-3770K clocked in at 3.5GHz, on the Scala.js test suite. This codebase contains 12,311 reachable methods.

The results are shown in Figure 4.5 and Table 4.4. We report the median running times, the speedup factor compared to sequential execution and the median absolute deviation for the running time. A decent amount of scaling can be observed, although it seems there is little to be gained beyond two threads. We believe bigger gains can be obtained with further work, replacing the simple usage of the Scala parallel collections by more targeted work stealing.

Performance of the Resulting Programs

It might be argued that the set of optimizations we have discussed and implemented are not representative of real-world optimizers. In Section 4.5.1, we will perform a detailed analysis of the impact of our optimizations on resulting programs. Speedups of 2–3x are common, and can go up to 7x.

# threads	time [ms]	speedup	MAD [ms]
1	4755.9	1.0	127.0
2	2689.1	1.8	194.8
4	2467.3	1.9	228.8
8	2465.7	1.9	169.8

Table 4.4 – Scaling with the number of threads – batch mode

4.2.7 Limitations

The approach presented in this paper focuses on speed for the incremental optimizer, and has some limitations.

Dynamic Language Features

Dynamic language features such as reflection and dynamic loading can quickly invalidate the assumptions made by our optimizer. However, this is not so much a limitation of the *incremental* nature of the optimizer as of the whole-program assumption. In fact, having an incremental optimizer can be beneficial if such language features are used, provided a Just-In-Time compiler can incrementally reoptimize parts of the program that are invalidated.

In the context of Scala.js, however, this is a non-issue. Scala.js does not provide any reflection nor dynamic loading by design. Dynamic loading is typically avoided in JavaScript environments, to enable bundling the entire application as a single JavaScript file. In larger applications, dynamic loading is used to improve startup times, but the whole program is still known before distribution; it is only fragmented after the fact (after whole-program optimizations) to enable lazy loading. Run-time reflection, on the other hand, is heavily used by dynamic languages, but is virtually never necessary in a language like Scala, which supports advanced compile-time metaprogramming features [7]. In practice, Scala.js developers replace run-time reflection by compile-time reflection for features such as automatic JSON serialization [46], statically typed RPC calls with the server [37], and so on.

Other Kinds of Whole-Program Optimizations

The optimizations we cover are essentially type-based. We have not yet considered other kinds of whole-program optimizations such as those requiring flow-based analyses. In theory, this is a severe limitation, but Section 4.5 will show that the optimizations we support are significant, and can bring idiomatic Scala.js code to a competitive level with respect to hand-written JavaScript code. The particulars are out of the scope of this dissertation, but the general insight is that we focus on generating code that is as friendly as possible to the next compiler in line, i.e., the JIT compiler. The optimizations we implement allow to remove the overhead of typical

Scala code, down to imperative, first-order monomorphic JavaScript code. JavaScript virtual machines are good at optimizing such code further using run-time profiling information.

In other contexts, the limitations currently imposed by our approach can be lifted with a two-staged process. During iterative development, only optimizations that lend themselves to being incrementalized are used. This already provides a major improvement over the non-optimized code, therefore reducing the time spent on compile/test cycles. We then add the other optimizations when emitting the final, production executable, to compensate for the limitations of the approach.

4.2.8 Related Work

Incremental reoptimization of generated programs was pioneered by Pollock and Soffa as early as 1985 [50]. This first work was concerned with optimizations strictly local to basic blocks. They later extended their algorithm [51] to accommodate procedure-level optimizations.

At about the same period, Cooper et al. [11] introduced a first framework for incremental re-compilation with interprocedural knowledge, which was extended later by Burke and Torczon [6]. Their framework tracks the assumptions made by the optimizer in the form of sets describing what callees *may* do without necessitating reoptimization. If any of the assumptions are broken, the procedure is marked for recompilation, ensuring the validity of the resulting program. However, their framework does not handle the reverse operation: should any new opportunity for optimizations arise, it is not detected. They also acknowledge the difficulty of recording all necessary assumptions. Knowledge queries can be viewed as a generalization of this idea with several improvements:

- They support non-boolean queries, for example, the set of possible targets of a dynamic call. This is critical to support object-oriented patterns, or simply inlining.

- They automatically record all the relevant assumptions, since the queries are the only interface between the optimizer and the program.

- They detect new opportunities for optimizations.

Chambers et al. [8] proposed an impressive framework that can be viewed as more general than our approach, since it applies to the entire compilation pipeline. Filtering nodes in their framework are basically equivalent to knowledge queries in terms of dependency tracking. However, as we explained in the Section 4.2.1, the measures and evaluations of this framework were focused on *accuracy*, omitting any discussion of the run-time overhead of their detection algorithm. Our approach has significant differences geared towards good run-time performance of the detection algorithm itself. In particular, it does not require factoring nodes, computed at run-time, to avoid memory blow-up, because we automatically compute the set of queries impacted by program changes instead of storing them in the dependency graph. Our algorithm also easily parallelizes, as we have shown in Section 4.2.5, which is an important

property nowadays. To achieve this, we sacrificed the fully automatic nature of the change detection algorithm. Since the evaluation of [8] only shows the number of methods that need to be reoptimized, rather than the actual run-time performance of their framework (e.g., how the incremental algorithm performs with respect to the batch algorithm), we cannot draw any measurable comparison with that work.

A related although somewhat different area is program analysis. There have been numerous works in that area in the past decades, including incremental whole-program analyses, e.g., [71, 55, 40, 67, 59]. The results of such advanced analyses can be used by optimizers, among other tools, to produce more efficent code. Incremental analyses can form a very powerful combination with incremental optimizers such as ours, since the whole pipeline can be incrementalized. Knowledge queries can be used to ask facts about the result of the static analysis. The changes to the static analysis results can be incrementally used to compute the set of knowledge queries that are impacted, which in turn will reoptimize only the appropriate set of methods. Actually, our program diffing algorithm can be viewed as a very weak form of incremental class hierarchy analysis. Combining state-of-the-art incremental program analyses with our optimizer could provide better results in the future.

4.2.9 Conclusion

We have presented a new approach to designing incremental whole-program optimizers using knowledge queries, and showed how to apply this approach to common optimizations in object-oriented and functional languages. Knowledge queries abstract away the details of the optimizer when analyzing changes to the program, and therefore create a modular interface between the optimization logic and the incremental logic.

The incremental whole-program optimizer of Scala.js, which uses this approach, shows speedups from 10x to 100x with respect to batch processing. This means that, in the broader context of the compilation pipeline, whole-program optimizations take negligible time.

There are, however, limitations to the technique, essentially because methods must be optimized independently. In other words, the optimization of a method may not depend on the result of optimizing other methods of the program. This prevents some advanced optimization decisions, e.g., inlining a target based on its optimized size, or optimizations that must be applied consistently in several methods or not at all. A work-around for this limitation is to apply additional, non-incremental optimizations only when producing the final executable, but not on every compile cycle.

Further work includes partially removing the above limitation, notably to enable scalar replacement of class fields, and integration with incremental program analysis techniques.

4.3 Encoding in JavaScript

After obtaining a linked, typechecked, and optimized IR, at some point we need to run that IR. Given how we specify the run-time semantics of the IR, as an extension of the ECMAScript specification, one natural implementation strategy would be to extend an existing JavaScript engine with direct support for interpreting the IR. This would certainly be entertaining—and might be the surest way to get top performance out of our IR—but would not get Scala.js any closer to running in actual web browsers or in stock Node.js processes. If we wanted to do this, the easiest path to get a performant implementation would probably be to build on top of Graal.js [70, 27].

Of course, since we want Scala.js to be runnable in existing JavaScript engines, we need to compile the IR down to ECMAScript code instead, which is the job of the Emitter. Most of this translation is straightforward compiler work. There are however a few interesting issues, which we briefly discuss in this section.

4.3.1 Expression-based to statement-based

The Scala.js IR is expression-based, with most of its construct being valid in an expression context, but JavaScript is statement-based. When translating the IR into JavaScript, we must somehow rewrite constructs in expression contexts that do not correspond to valid JavaScript expressions. As a very simple example, consider the following snippet of pseudo-JavaScript where blocks are valid expressions:

```
obj.meth({
  const x = foo(42);
  x*x
});
```

One easy way to rewrite the above snippet into valid JavaScript code, which is used by some compilers, is to wrap every non-expression used in an expression context into an IIFE (Immediately Invoked Function Expression), adding a `return` at the end:

```
obj.meth((function() {
  const x = foo(42);
  return x*x;
})());
```

Such a strategy allows the compiler to stay extremely simple, but does not produce the most efficient code. The Scala.js emitter, like most non-trivial compilers to JavaScript, uses a different strategy involving unnesting of non-expressions. Since the argument to `obj.meth` must be a JavaScript expression, we simply extract it in a temporary variable, which is the unnest step:

```
const temp1 = {
  const x = foo(42);
  x*x
};
obj.meth(temp1);
```

This does not completely solve the problem, since a block is still used as the right-hand-side of an assignment. We can however "push" the assignment into the block, as follows. We must turn the `const` into a `let` to do that, but this has no effect on the run-time semantics.

```
let temp1;
{
  const x = foo(42);
  temp1 = x*x;
};
obj.meth(temp1);
```

Since it is always possible to extract non-expressions into temporary variables, and since we can always push an assignment into complex constructs, we can compile away all non-expressions in expression context.

Besides blocks, assignments can be pushed inside `if`, `switch`, `try..catch`, `try..finally` and labeled blocks. For the latter, every `return` from the block is replaced by an assignment and a `break`. For example, the IR snippet

```
val y: T = label[T]: {
  ...
  if (x)
    return@label someT;
  ...
  someOtherT
}
```

is compiled as:

```
let y;
label: {
  ...
  if (x)
    { y = someT; break label; }
  ...
  y = someOtherT;
};
```

4.3.2   Exotic behavior of Scala objects

The run-time semantics of the IR specify that Scala objects are exotic objects, which contributes to the Closed World property we saw in 3.3.2. Recall that exotic objects in ECMAScript are objects whose *internal methods* do not (all) behave as specified for ordinary objects. If

Scala objects were ordinary objects, we could simply translate Scala classes to ECMAScript classes.

First, note that most of the exotic behavior of Scala objects is to trigger Undefined Behavior. For example, trying to set the value of a field that is not exported, through the [[Set]] internal method, is undefined behavior. In order to actually detect the erroneous behavior and throw an exception for any possible key, we would need an actually exotic behavior. For example, we could implement the detection by wrapping every Scala object in a Proxy, using an appropriate `"set"` handler. However, doing so would most likely have dramatic consequences on the run-time performance of Scala.js code, which would not be acceptable even in development mode. In practice, for the exotic behaviors that trigger undefined behavior, we simply leverage the property of undefined behavior that anything can happen, and let it go: we allow the [[Set]] internal method to go through.

Recall from Section 2.1.6 that we argued that undefined behaviors should be caught in development mode so that erroneous code does not accidentally work. This is one area where we break our own rule. Ideally, in some future version of Scala.js, we should provide the option to generate all the Proxies necessary to properly catch all undefined behaviors.

Besides the undefined behaviors, the other exotic behaviors can be implemented relatively easily:

- the behavior of [[Get]] and [[Set]] for exported properties is implemented using ES properties with getters and setters,

- for exported methods, the convoluted behavior of [[Get]] simply translates into a regular instance method, and

- internal state and methods that are supposed to be private to the IR are implemented using regular ES fields and methods, with the precaution that their name must never coincide with the names of exported properties or methods.

4.3.3 Characters

The Scala.js IR features a primitive `char` type for characters. From an observable semantics point of view, they are specified to be objects whose `toString()` method returns a one-code-unit `string` with that character inside. However, internally, `char`s are often used as primitives, and used in computations by repeatedly converting them to and from `int`s. It would be a performance issue should each such operation box and/or unbox the primitive numbers inside objects.

In order to avoid the boxing overhead, expressions whose static type is `char` are compiled down as primitive numbers in the range $[0, 2^{16} - 1]$. That gives the wrong behavior for `toString()` as well as other operations of `any`, such as `is/asInstanceOf`, though, when a `char` is upcast

to `any`. The Emitter therefore inserts box operations from primitive numbers to instances of a `Char` class when a primitive `char` is upcast to `java.lang.Character` or above, and unbox operations for downcasts. Basically, we use *coercion semantics* [49, §15.6] for the apparent subtyping relationship between `char` and `java.lang.Character`.

4.3.4   Longs

The other primitive type that does not straightforwardly map to JavaScript is `long`, which is a signed 64-bit integer. Such a type is probablematic from a performance point of view, because it cannot be represented in any primitive JavaScript type, and operations must be implemented using costly user-space functions. Longs in Scala.js used to be the one feature for which we would receive performance complaints. So much that we dedicated significant effort into optimizing them beyond the state of the art. Our implementation of 64-bit integers will be discussed in detail in Section 4.4.

4.3.5   Overloaded constructors

The Scala.js IR, just like Scala, Java and the JVM bytecode, supports compile-time overloads for constructors. While for methods, we can mangle them in JavaScript to maintain the overloading behavior, we cannot create multiple constructors for a JavaScript class. This is easily solved by encoding IR constructors as regular methods in JavaScript, and converting each `new` from the IR into a JS `new` chained with a call to the constructor method. For example, the following IR snippet:

```
class A extends Object {
  val x: int
  def init___I(x: int) {
    this.Object::init___();
    this.x = x;
  }
  def init___() {
    this.A::init___I(5)
  }
}
val a: A = new A.init___I(6)
```

would be compiled to:

```
class A extends Object {
  constructor() {
    super();
    this.x = 0; // the zero of the type int
  }
  init___I(x) {
    Object.prototype.init___.call(this);
    this.x = x;
```

```
      return this;
  }
  init___() {
    A.prototype.init___I.call(this, 5);
    return this;
  }
}
const a = new A().init___I(6);
```

Every constructor method performs `return this` so that it can be chained at the instantiation site. The JavaScript constructor for a Scala class is only responsible for initializing all the fields of the instance to the zero of their respective types.

As an optimization, if a class has only one constructor, and is never inherited, we can inline its unique constructor method inside the JavaScript constructor. For example, if the above class did not have its second constructor `init___`, Scala.js would compile it as:

```
class A extends Object {
  constructor(x) {
    super();
    this.x = 0; // the zero of the type int
    // inlined content of init___I
    Object.prototype.init___.call(this);
    this.x = x;
    // 'return this' is not necessary
  }
}
const a = new A(6);
```

4.3.6 References to JS global variables

The IR features one kind of node which causes unexpected difficulties, namely accessing a variable from the JS global scope. The abstract syntax for that node is

`global:x`

where x is a valid JS identifier. The run-time semantics specification mandates that this expression look up the identifier x in the lexical scope where the Scala.js program was introduced, which often is the global scope, or Global Environment Record to be precise.[7] A naive compilation scheme would simply splice the identifier x in the generated JavaScript code. However, this is incorrect, as it can expose compiler artifacts if the compiler uses the identifier x for its own purposes (as a temporary variable, or as the variable used to store a top-level class value, etc.).

In order to avoid this issue, either the x in the global reference or the compiler artifact must be renamed. The former would alter the semantics, so the only real option is the latter. This can

---

[7]In some special cases, it can be a different scope. For example, in a CommonJS module for Node.js, it would be the *module scope*, containing magic variables such as `__filename` and `require`.

have consequences at a local scale (for example, clash with a local variable) or at a global scale (for example, clash with a global store for a class value). Analyzing the entire program to find references to global variables, which must not be used for compiler internals, would be too expensive, especially in incremental runs of the emitter.

Therefore, we take a two-layered, two-stage optimistic approach. The two stages are:

1. optimistically emit everything assuming there will not be any clash, and record the local and global variables that are used, and

2. if there was a clash, rerun the emission, but this time actively avoid clashes (the compiler now has the list of global variables to avoid).

The two layers are:

- in a method, allow the compiler to use any identifier as artifacts, and record all accesses to global variables, and

- outside methods (for class values, etc.), only allow the compiler to use identifiers starting with $ (and with length > 1) as artifacts, and only record accesses to global variables of that shape.

This allows references to global variables that are "normal" identifiers (not starting with $) not to pollute the entire program, to avoid spurious recompilation of the whole program, while at the same time correctly avoiding all clashes. Based on anecdotal evidence in our own test suite, we expect this layered approach to cause very few clashes in the first place, allowing the emitter to run in a single-pass for the whole program most of the time, and only using the two local passes for a few methods. To further improve this, the results of the global tracking are cached between incremental runs of the emitter, so that subsequent runs can avoid the second pass on the whole program in most cases even for codebases that would normally require one.

4.3.7 The meta-object protocol

As we saw in Section 2.1.3, there is some amount of run-time reflection that needs to be supported by Scala.js. Most are implemented in user-space inside `java.lang.Class`, on top of the information provided by the run-time semantics (via the abstract operation `SJSIRClassOf` in Section 2.6). The others are `getClass()` and `is/asInstanceOf`, which have dedicated expressions in the IR. All of those operations essentially rely on the data stored in SJSIR Classes, as defined in Section 1.1 of the run-time semantics. Since they need to be available at run-time, it is necessary to partially reify SJSIR Classes in the generated JavaScript code, which is simply done as instances of an internal class `TypeData`. That class and its instances are protected from tampering by Scala.js user code or external JavaScript code by scoping. Similarly, the [[SJSIR-Class]] internal slot of Scala objects is reified as a special field on the `prototype` of JavaScript

classes encoding the Scala classes. This special field is used to implement `getClass()` and `is/asInstanceOf`.

4.3.8 Arrays

On the JVM, and by extension in the Scala.js IR, arrays preserve the type of their elements at run-time, while JavaScript arrays do not. Scala.js IR arrays are therefore implemented as wrappers on top of a JavaScript array, with the special field encoding the [[SJSIRClass]] internal slot.

4.4 64-bit integers

Scala.js and its IR, among plenty of other languages, feature fixed-width integer data types, including `Int`s and `Long`s, which are 32 and 64 bits wide, respectively. However, the JavaScript language only features `Double`s, i.e., 64-bit floating-point numbers, which means that compilers need to encode the semantics of integer data types. Since the advent of asm.js [3], there exists a well-known, efficient encoding of signed 32-bit integers with wrapping operations. For example, the operation `a + b` for `Int`s can be encoded as `(a + b) | 0`. The encoding, which we describe in Section 4.4.1, has become so widespread that JavaScript VMs optimize it away.

There is no such encoding for 64-bit integers, however. The state-of-the-art technique consists in storing `Long`s as instances of a class provided by the runtime of the language. Each primitive operation is then implemented in a method and allocates a new instance for the result. For example, a possible implementation of bitwise | in ECMAScript 2015 would be the following:

```
class RuntimeLong {
  constructor(lo, hi) {
    this.lo = lo;
    this.hi = hi;
  }

  or(b) {
    return new RuntimeLong(this.lo | b.lo, this.hi | b.hi);
  }
}
```

Even though a bitwise | is easy to implement with this representation, the same cannot be said of most other operations, including simple arithmetic operations such as addition. Moreover, new instances of `RuntimeLong` are allocated for every primitive operation. Note that textbook algorithms for multi-precision arithmetics, such as described in [31], cannot be used, since they assume that single-precision instructions provide certain "services" to multi-precision arithmetics (e.g, `adc`–add with carry–can reuse the carry of a previous addition).

Such encodings cause significant performance issues on JavaScript engines [36, 66]. `Long` operations are typically one to two orders of magnitude slower than their `Int` counterparts, as we show in Section 4.4.3. These prohibitive numbers have even led some languages to entirely disregard correctness, and compile their `Long`s as simple `Double`s, which only provide 53 out of the 64 bits of precision for integer values. This includes languages specifically designed to cross-compile to JavaScript, such as Ceylon [54]. This is a drastic trade-off to make, as primitive values do not behave consistently across platforms, potentially leading to bugs that are difficult to track.

To ensure portability, the Scala.js language and compiler have always chosen the correct 64-bit semantics first, leading to a slow implementation of `Long`s. They have been responsible for most—if not all—major performance issues in Scala.js applications, where users reported a surprisingly inefficient behavior (compared to the JVM version, or to subjective expectations, for example). Anecdotally, rewriting the implementation of `java.util.Random` from the specified algorithms using `Long`s to a specialized, hand-optimized one using `Int`s and `Double`s (though observably equivalent) turned a user's program from being unusably slow to being very responsive. The implementations of `BigInteger` and `BigDecimal`, which use `Long`s under the hood, have also been repeatedly reported to be excessively slow. Those issues were strong motivation for providing efficient `Long`s in Scala.js, which we address in this section.

First, we survey in Section 4.4.1 the existing implementations of `Long`s in GWT [25], TeaVM [62] and Kotlin [33] (the only three open-source ahead-of-time compilers to JavaScript with correct `Long`s). Through a set of micro benchmarks, we identify what is the best implementation for each operation, and we combine those into a best-of-breed implementation of `RuntimeLong` for Scala.js.

Second, using stepwise refinements, we improve upon this implementation in Section 4.4.2, yielding measurable improvements on virtually all operations, including an order of magnitude improvement on division, modulo and `toString`. The implementation obtained at this point can be readily reused by GWT, TeaVM, Kotlin and Doppio.

Thirdly, in Section 4.4.3 we demonstrate how one can achieve even better performance by leveraging our optimizer from Section 4.2, or any other optimizer for a statically typed language. We exploit the fact that the implementations of all operations except `/`, `%` and `toString` (which we do not improve any further) are very compact. Our implementation of `RuntimeLong` greatly benefits from standard optimization techniques, i.e., inlining, scalar replacement (with a twist) and integer rewrite rules. This brings an order of magnitude improvement to most operations as well as to an implementation of SHA-512.

We evaluate our implementation against that of GWT, TeaVM and Kotlin. Results show speedups from 3.6x to 60x on a SHA-512 benchmark, and 3x to 20x on individual operations. Furthermore, we demonstrate that, barring division and remainder, operations on `Long`s are only 3x slower than those on `Int`s on Chrome, and 5x slower on Firefox. The latter

result has an important consequence: it is now possible for statically typed languages compiling to JavaScript to have both fast and correct `Longs`, without having to sacrifice one property to the other.

Finally, in Section 4.4.4, we mechanically verify the validity of user-land operations using Predicate-Qualified Types: comparisons, bitwise operators, as well as all arithmetic operators except `*`, `/` and `%`.

The work presented in this section has been done in collaboration with Georg Schmid.

4.4.1 Survey of existing implementations

We begin our quest for fast 64-bit integers on JavaScript with a survey of the existing implementations. To the best of our knowledge, there are four open-source compilers providing correct `Longs` besides that of Scala.js: those of Doppio [66], GWT [25], TeaVM [62] and Kotlin [33]. Doppio and Kotlin share a common implementation derived from the Google Closure Library [24]. Since Doppio stands apart as a JVM implementation in JavaScript rather than an ahead-of-time compiler to JavaScript, we discard it in favor of Kotlin. We therefore focus our analysis on GWT, TeaVM and Kotlin, whose `Long` implementations can be found at [23, 63, 34].

All benchmarks and discussions of TeaVM include a simple performance "fix" that we have applied to the multiplication algorithm. The original algorithm normalizes inputs to positive values, which is superfluous, as multiplication in 2's complement produces the same results whether the bit pattern is interpreted as signed or unsigned. Removing the normalization brings a 22% performance improvement to the TeaVM multiplication.

Background: JavaScript and its numbers

Before looking at the various implementations, it is important to understand what JavaScript gives us. Specification-wise, JavaScript only has 64-bit double precision floating point numbers, i.e., `Doubles`. All arithmetic operations (`+ - * / %`) have `Double` semantics. However, JavaScript also provides 32-bit integer bitwise operators. Their operands are technically `Doubles`, but before applying the operation, the operands are coerced to signed 32-bit integers (wrapping, rather than capping, their values modulo 2^{32}). The result is then converted back to a `Double`. The available bitwise operations are: and `&`, or `|`, xor `^`, shift left `<<`, arithmetic shift right `>>` and logical shift right `>>>`.[8]

We can use these operators, and in particular `|`, to concisely implement signed 32-bit *arithmetic* operations. The fact that `x|0` forces wrapping `x` around signed 32 bits allows us to implement `+ - / %` easily as `(a+b)|0`, `(a-b)|0`, `(a/b)|0` and `(a%b)|0`, respectively. Signed

---

[8]Technically, `>>>` works on *unsigned* 32-bit integers, but we will ignore this detail as we can always "fix" it by using `(a >>> b)|0` instead of `a >>> b`.

32-bit multiplication is harder. `(a*b)|0` does not work because the intermediate result `a*b` might already loose precision in the 11 least significant bits. Fortunately, ECMAScript 2015 provides signed 32-bit multiplication under the builtin `Math.imul(a, b)`, and there exists a polyfill for older versions of JavaScript.

The above encoding of signed 32-bit arithmetic operations was made popular by `asm.js`. Since then, JavaScript VMs have started optimizing the `(a+b)|0` idiom so that they essentially amount to "primitive" signed 32-bit operations, paradoxically making them slightly faster than the corresponding double operations such as `a+b`. This allows to efficiently implement `Ints` and their operations in JavaScript.

For all intents and purposes, JavaScript therefore gives us *two* efficient numeric types: `Doubles` and `Ints`. We can build on them to implement `Longs`. It is also worth mentioning that `Doubles`, by their nature, provide accurate integer values up to 2^{53} (in absolute value). They have an effective precision of 53 bits in addition to a sign bit.

Overview of the design choices

We briefly survey the design decisions made in three prior implementations.

GWT   Unlike the other implementations, which use 2 `Ints`, GWT represents its `Longs` using three `Ints`, `l`, `m` and `h`, storing bits 0–21, 22–43 and 44–63, respectively. The motivation for this design choice is that components can be manipulated without incurring overflow in intermediate results. For example, the implementation of addition (in Java) is:

```java
public static LongEmul add(LongEmul a, LongEmul b) {
  int sum0 = a.l + b.l;
  int sum1 = a.m + b.m + (sum0 >> BITS);
  int sum2 = a.h + b.h + (sum1 >> BITS);
  return create(sum0 & MASK, sum1 & MASK, sum2 & MASK_2);
}
```

The carries are part of the intermediate results `sum0` and `sum1`, and can easily be propagated by shifting them. A direct benefit is that `add` is branchless, at least at the JavaScript source code level.

This design decision must be placed in the context in which GWT was first developed. The implementation of `Longs` in GWT is ancient, and predates the `asm.js` encoding of 32-bit integers. As a matter of fact, GWT does not correctly implement `Ints`, but rather as plain JavaScript numbers, which basically means `Doubles`. Therefore, the implementation of `Longs` cannot rely on the 32-bit wrapping semantics of primitive `Ints`. Besides additive operators, however, the encoding with 3 numbers is counterproductive. For example, bitwise and comparison operators need to handle 3 pairs of operands rather than 2.

TeaVM and Kotlin   Unlike GWT, which implements its `LongEmul` class in Java, TeaVM and Kotlin manually implement their Long class in JavaScript. They use a more natural encoding of `Longs`, using a pair of `Ints` each holding 32 bits. This design tries to better leverage the optimizations of contemporary JavaScript engines, which are good at dealing with 32-bit integers. For example, TeaVM exploits the `asm.js` encoding to efficiently implement `inc` (+1) on Longs.

```
function Long_inc(a) {
  var lo = (a.lo + 1) | 0;
  var hi = a.hi;
  if (lo === 0)
    hi = (hi + 1) | 0;
  return new Long(lo, hi);
}
```

Note the two occurrences of `(x+1)|0`, which implement wrapping 32-bit additions.

This leads to more complicated implementations of additive operators, however, as operands need to be decomposed in chunks of 16 bits to prevent overflows in intermediate operations to lose carries. To compensate, TeaVM chooses to use fast paths when operands actually fit in 32 bits or 53 bits, leveraging efficient primitive operations on integers and doubles. For example, the addition in TeaVM is as follows:

```
function Long_add(a, b) {
  // Fast paths
  if (a.hi === (a.lo >> 31) && b.hi === (b.lo >> 31))
    return Long_fromNumber(a.lo + b.lo);
  if (Math.abs(a.hi) < MAX_NORMAL && Math.abs(b.hi) < MAX_NORMAL)
    return Long_fromNumber(Long_toNumber(a) + Long_toNumber(b));

  // Slow path (omitted)
  return new Long(...);
}
```

Interestingly, Kotlin has a similar-looking implementation of addition, but does not include fast-paths for small values.

Evaluation of existing implementations

In order to determine which implementation is best for each operation, we performed a set of micro-benchmarks on representative operations. Each operation is performed on a dataset of 100x100 operands, for a total of 10,000 iterations. The datasets have been randomly generated using a uniform distribution of values fitting in a certain number of bits in 2's complement representation.

Table 4.5 lists all our micro-benchmarks. `longN` represents a randomly generated N-bit value, sign-extended to a Long. N and M can be 32, 53 or 64. Some implementations take shortcuts

Name	Description
Nop	Baseline for micro-benchmarks, without any specific Long operation
Xor	`long64 ^ long64`
Add	`long64 + long64`
Mul	`long64 * long64`
Div N/M	`longN / longM`
Div N/pow2	`longN / longPow2`
ToString N	`Long.toString(longN).length()`

Table 4.5 – List of the micro-benchmarks

when their operands fit in 32 or 53 bits. M can also be 8, providing test cases where the divisor is significantly smaller than the dividend, which typically trigger worst-case behaviors of the division algorithms. Similarly, `longPow2` is a dataset of randomly generated Long values `v = 1L << n` for $n \in [0, 63]$, for which some division algorithms use `>>>` as an optimization. In all cases, values are retrieved dynamically from an array, which means optimizers cannot perform ahead-of-time simplifications. The results of all operations are xor'ed together, and the final result is checked for correctness to prevent optimizers from eliminating the computations as dead code.

We run the benchmarks on Linux 4.4.0-93 on an Intel® Core™ i7-4790 CPU clocked at 3.60GHz, with two browsers: Chrome 57.0.2987.110 and Firefox 55.0.2. The versions of TeaVM and Kotlin are 0.5.1 and 1.1.4-3, respectively. We use the "ADVANCED" optimization level of TeaVM rather than "FULL", because the latter yielded significantly worse results on Firefox. For GWT, we considered versions 2.7.0 and 2.8.x, but chose the former because it was overall faster for those benchmarks. Figures 4.6, 4.7 and 4.8 show the results. They include a fourth implementation, Scala.js "best of breed", which we discuss in Section 4.4.1. Since the SEMs (Standard Error of the Mean) of all our measurements are at least 2 orders of magnitude smaller than their respective means, we do not clutter the graphs with error bars.

Xor   TeaVM is by far the best implementation of `xor` on Chrome, and on par with Kotlin on Firefox. GWT is unsurprisingly the slowest, having to deal with 3 fields rather than 2. In general, GWT performs poorly on all bitwise operations, because its data representation does not provide any shortcuts in these cases.

Addition   We would have expected GWT to shine in the addition case, given that its design allows for a branchless and compact implementation of addition. However, it happens to be the slowest. The fact that Kotlin wins over TeaVM on this benchmark is due to its lack of fast-path for small values. Indeed, we use random 64-bit operands, most of which should not fit in a double. In that sense, this benchmark is not fair to TeaVM. If we remove the fast-paths from TeaVM's algorithm, the two implementations become indistinguishable.
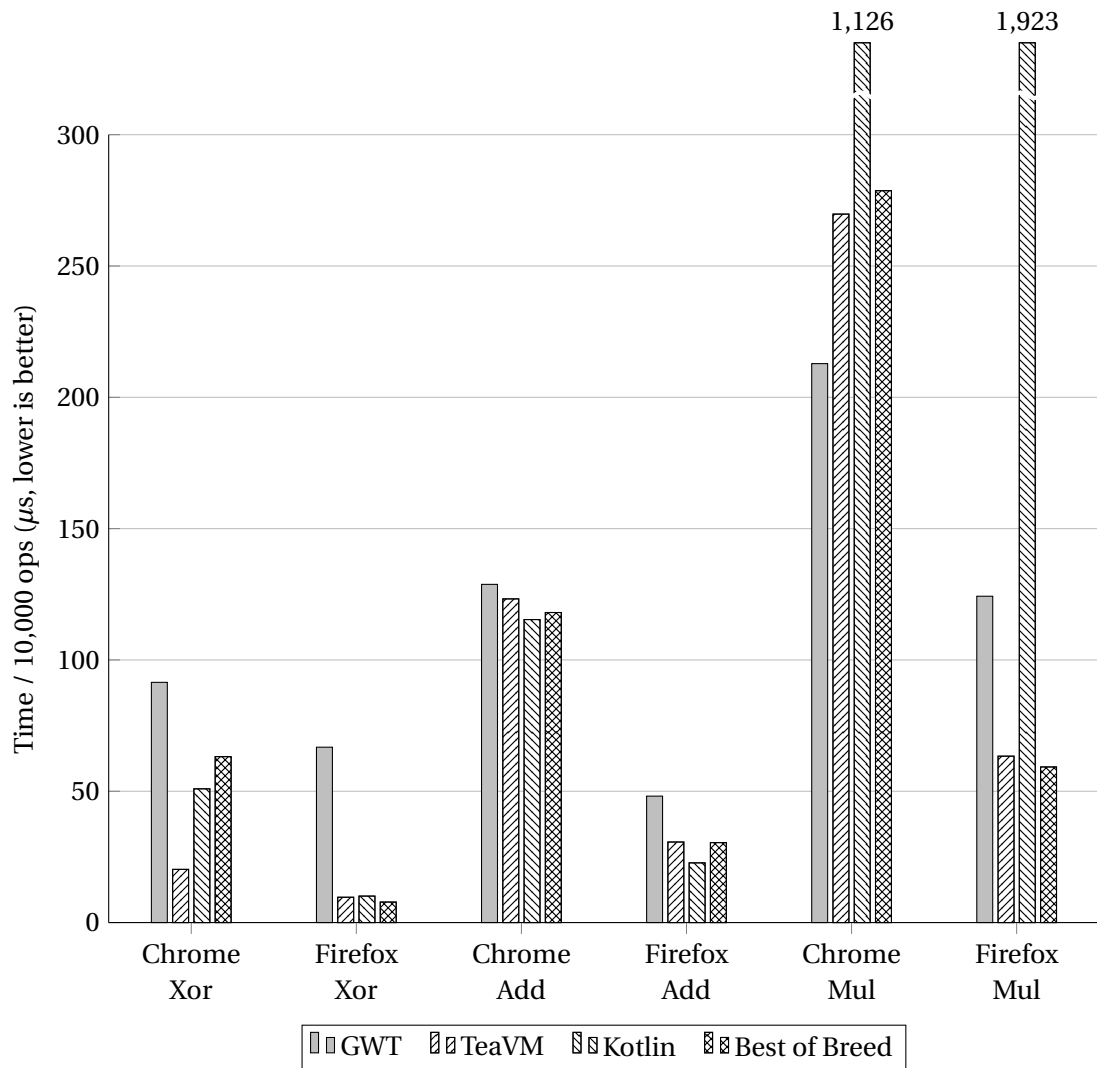
Figure 4.6 – Micro-benchmarks of existing implementations for Xor, Addition and Multiplication
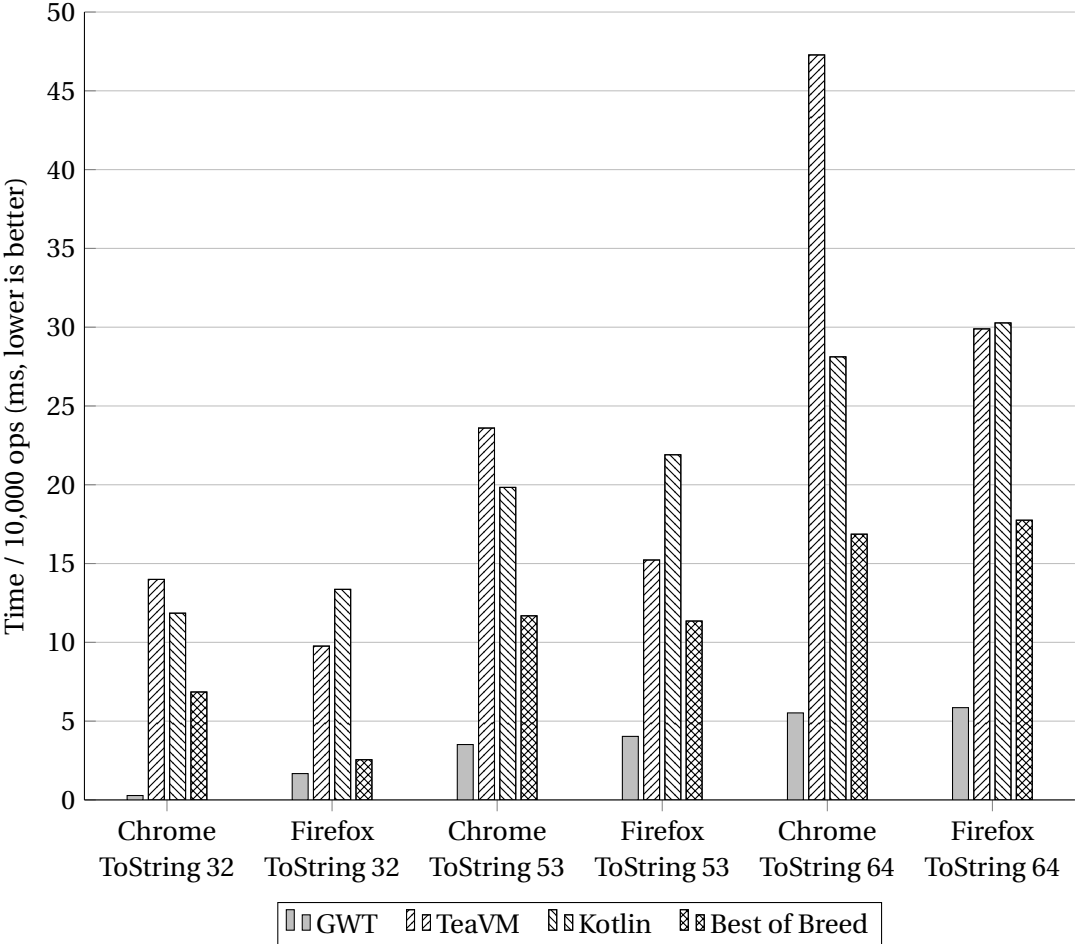
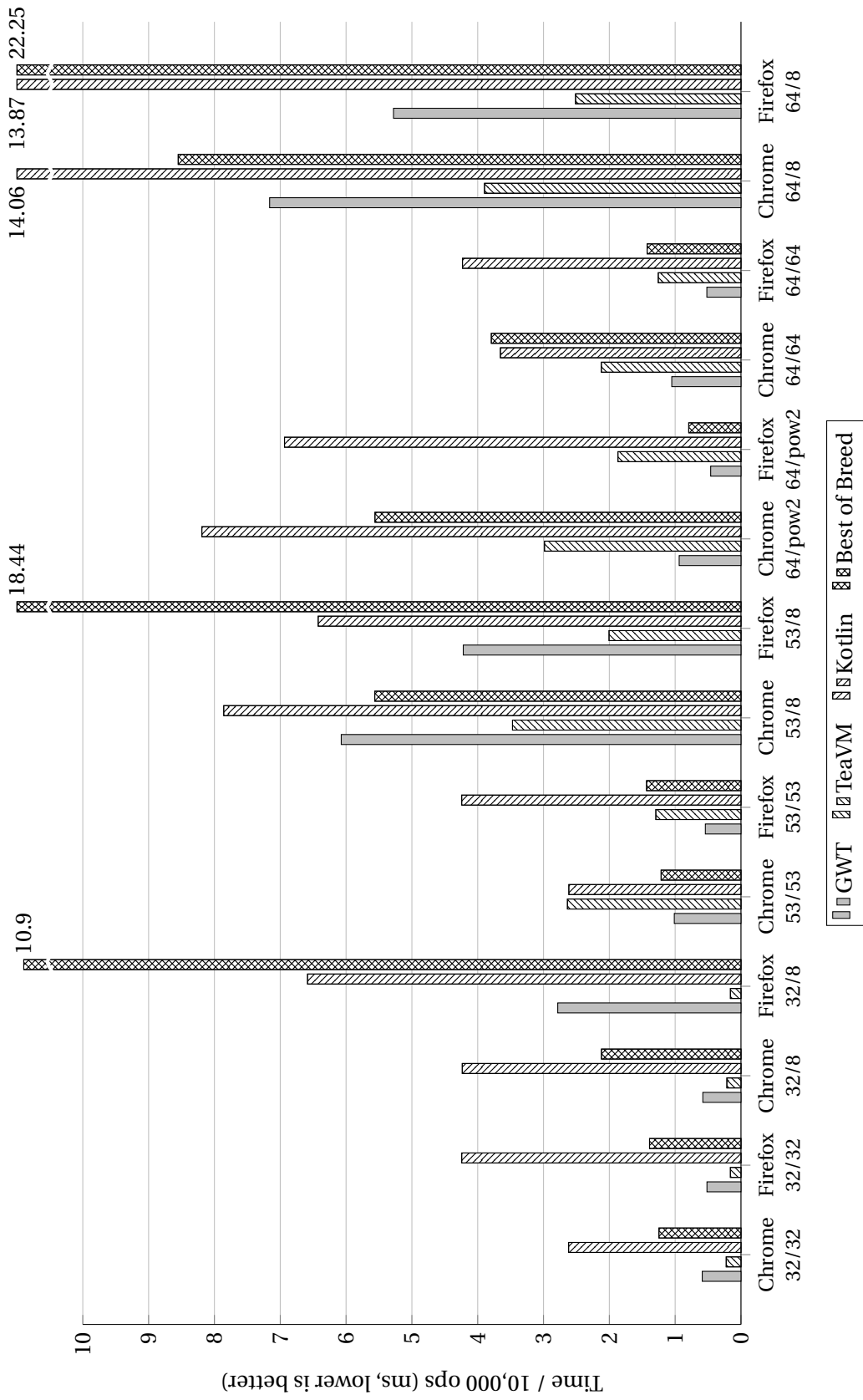Figure 4.7 – Micro-benchmarks of existing implementations for `toString`

Figure 4.8 – Micro-benchmarks of existing implementations for division

Multiplication GWT exhibits the best implementation of multiplication on Chrome, with TeaVM a close second. The latter is however twice as fast as GWT on Firefox.

Division TeaVM and GWT share the podium for division. GWT gives better results when the divisor has the same magnitude as the dividend, whereas TeaVM shines with small divisors. GWT also has a special optimization when the divisor is an exact power of 2, which gives it excellent results in those cases.

Conversion to string GWT's `toString` outshines the other implementations. This is easily explained by inspection of their algorithm. They divide by 10^9 at each step, leveraging JavaScript's primitive conversion of `Int` to string, whereas TeaVM and Kotlin use steps of 10^6 and 10, respectively.

Best-of-breed implementation for Scala.js

Using the knowledge gathered in the previous section, we can write an initial implementation of `RuntimeLong` for Scala.js, with the best implementation of each operation. We choose the data representation with two integers, since TeaVM won on most non-division operations, and GWT's division algorithm does not directly depend on the data representation. The operations are picked from GWT and TeaVM:

- We obviously use GWT's implementation of `toString`.

- For division, there is no best implementation. We choose GWT's on a leap of faith that we will be able to improve its behavior with small divisors. TeaVM's implementation relies on a separate implementation of unsigned 80-bit integers, which we would like to avoid for code size reasons.

- For the other operations, we use TeaVM's implementations, without the fast-paths for small values. We will make the fast-paths irrelevant in Section 4.4.2 anyway.

The graphs of the previous section compare this implementation with those of GWT, TeaVM and Kotlin. We would expect it to be better than all the alternatives, but this is not the case. For xor, addition and multiplication, it exhibits similar performance characteristics to the best existing implementations, with a notable exception for xor on Chrome. Our (unproved) hypothesis for our implementation being slower is that Scala.js uses instance methods (e.g., `a.xor(b)`) rather than top-level functions (e.g., `Long_xor(a, b)`), and that JavaScript JITs need more expensive deoptimization guards for instance methods. We could improve this in Scala.js, but the issue will become moot in Section 4.4.3, once we have our own optimizer inline those functions. Another possibility is that the Scala.js compiler is overall of lesser quality than the others, although our biased mind is reluctant to consider it.

Division is overall 3x slower than GWT, our chosen reference. This is easily explained because we took some shortcuts when porting GWT's division algorithm. More specifically, we kept `RuntimeLong` as an immutable class, forcing us to allocate more instances than GWT, which mutates intermediate values in-place. Consequently, `toString`, which builds on division, is also overall 3x slower than GWT. The `toString 32` benchmark on Chrome exhibits an abnormal 25x slowdown which we did not manage to explain.

4.4.2   Improvements to `RuntimeLong`

Now that we have a baseline implementation in Scala.js, built from the best-in-class existing implementations, we can start optimizing it. These optimizations are in general independent of the source language and its compiler, which means that they can be readily ported to GWT, TeaVM and Kotlin to improve the performance of their `Long`s. There are two general themes that drive our optimizations:

- Reducing the number of branches, which is a typical trick in micro-optimization both for performance and code size, and

- Exploiting the properties of the 2's complement representation, in particular the relationships between the signed and unsigned interpretations of bit patterns.

Figures 4.9, 4.10 and 4.11 show the results of benchmarks for the various refinements. The improved algorithms derived in this section are shown as "User-land". "With scalar replacement" and "With integer simplifications" refer to further optimizations discussed in Section 4.4.3.

Removing branches: shifts

As a simple illustration of the theme of removing branches, we show the transformation we apply on <<, i.e., shift left. It is worth pointing out that in Java, Scala and JavaScript, shift operators mask their right-hand-side to the 5 least significant bits for `Int`s (`rhs & 31`) and the 6 least significant bits for `Long`s (`rhs & 63`). The base implementation coming from TeaVM is as follows:

```
def <<(n: Int): RuntimeLong = {
  val n1 = n & 63
  if (n1 == 0)
    this
  else if (n1 < 32)
    new RuntimeLong(lo << n1, (lo >>> (32 - n1)) | (hi << n1))
  else if (n1 == 32)
    new RuntimeLong(0, lo)
  else
    new RuntimeLong(0, lo << (n1 - 32))
}
```
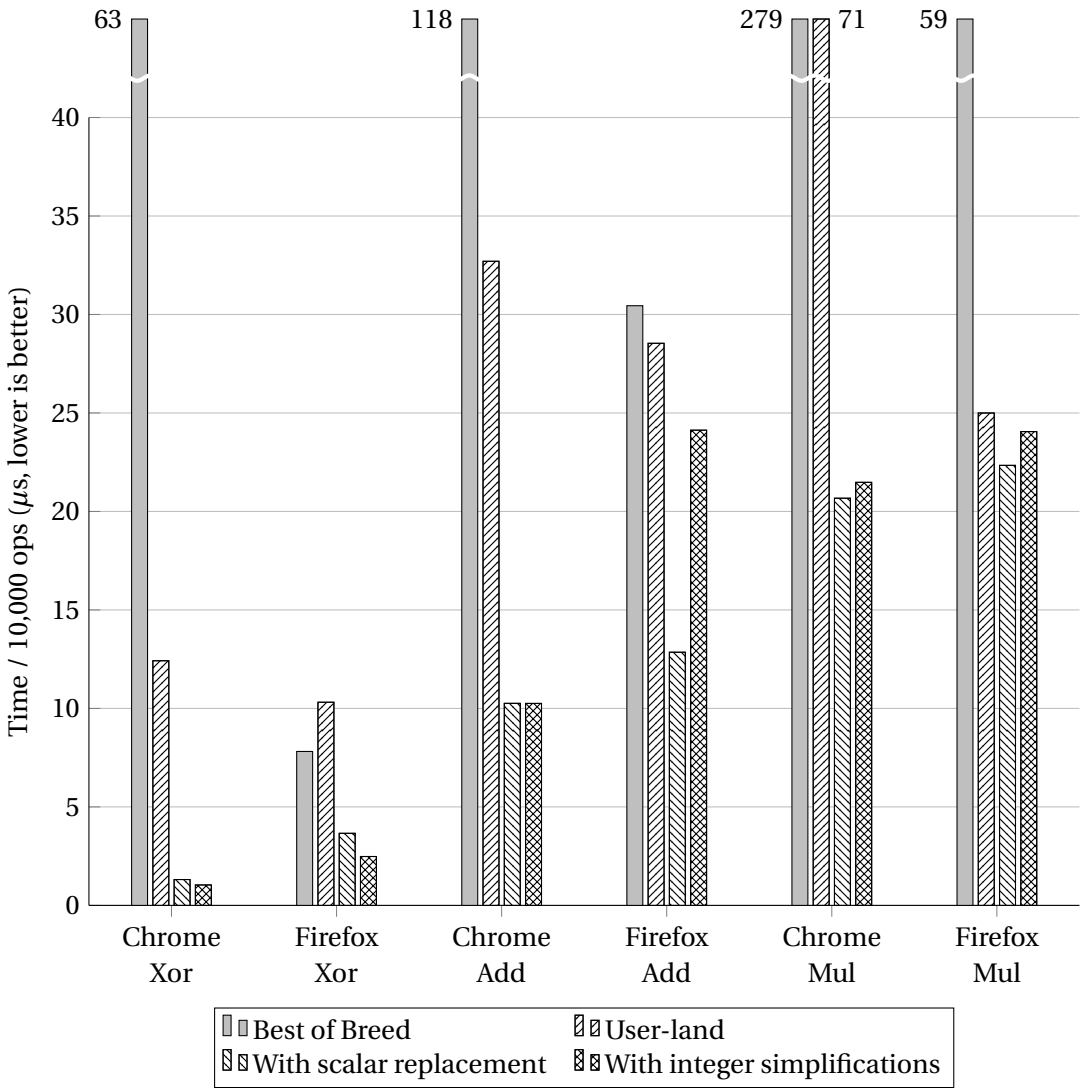
Figure 4.9 – Micro-benchmarks of our improvements to Xor, Addition and Multiplication
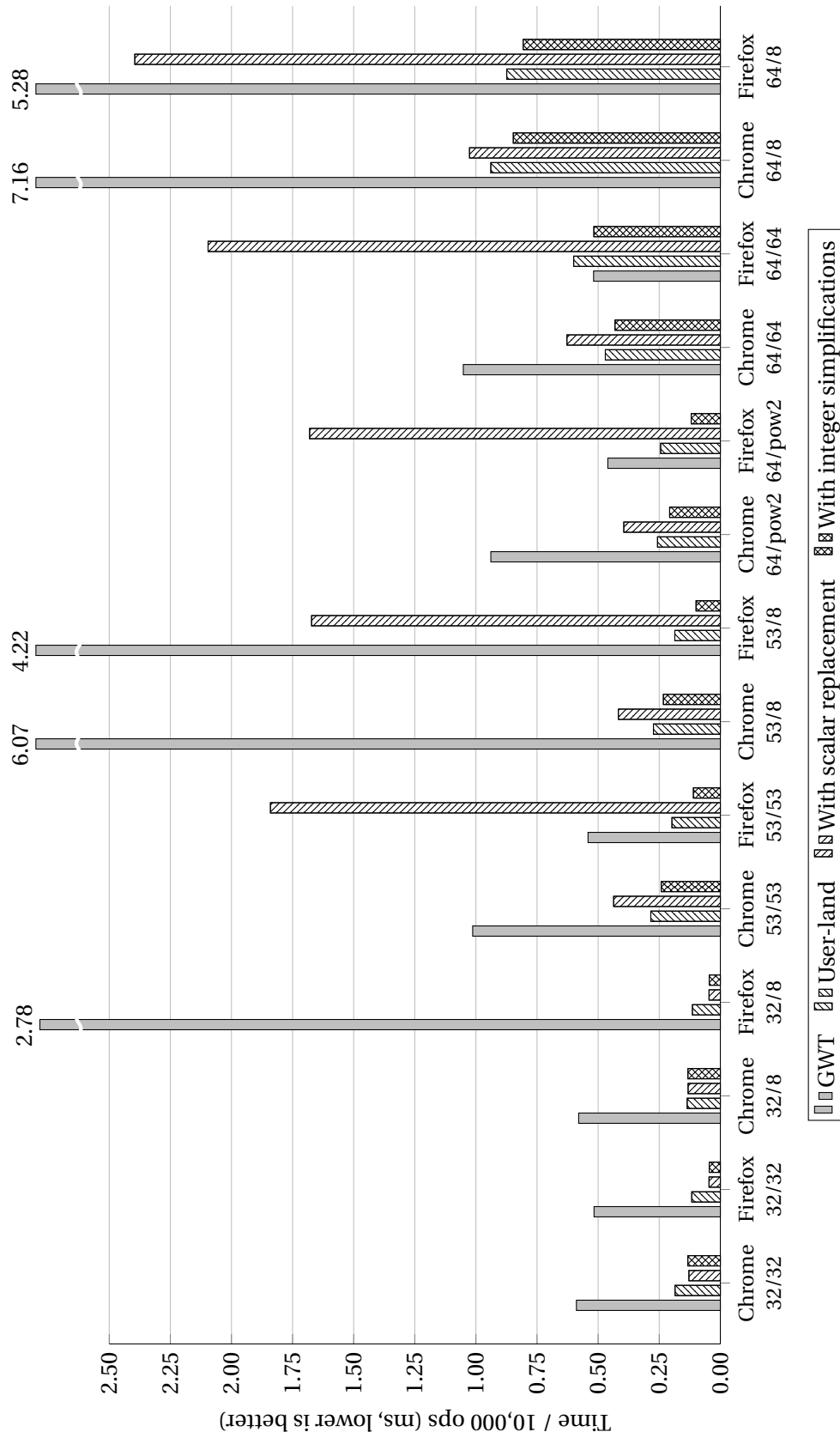
Figure 4.10 – Micro-benchmarks of our improvements to division
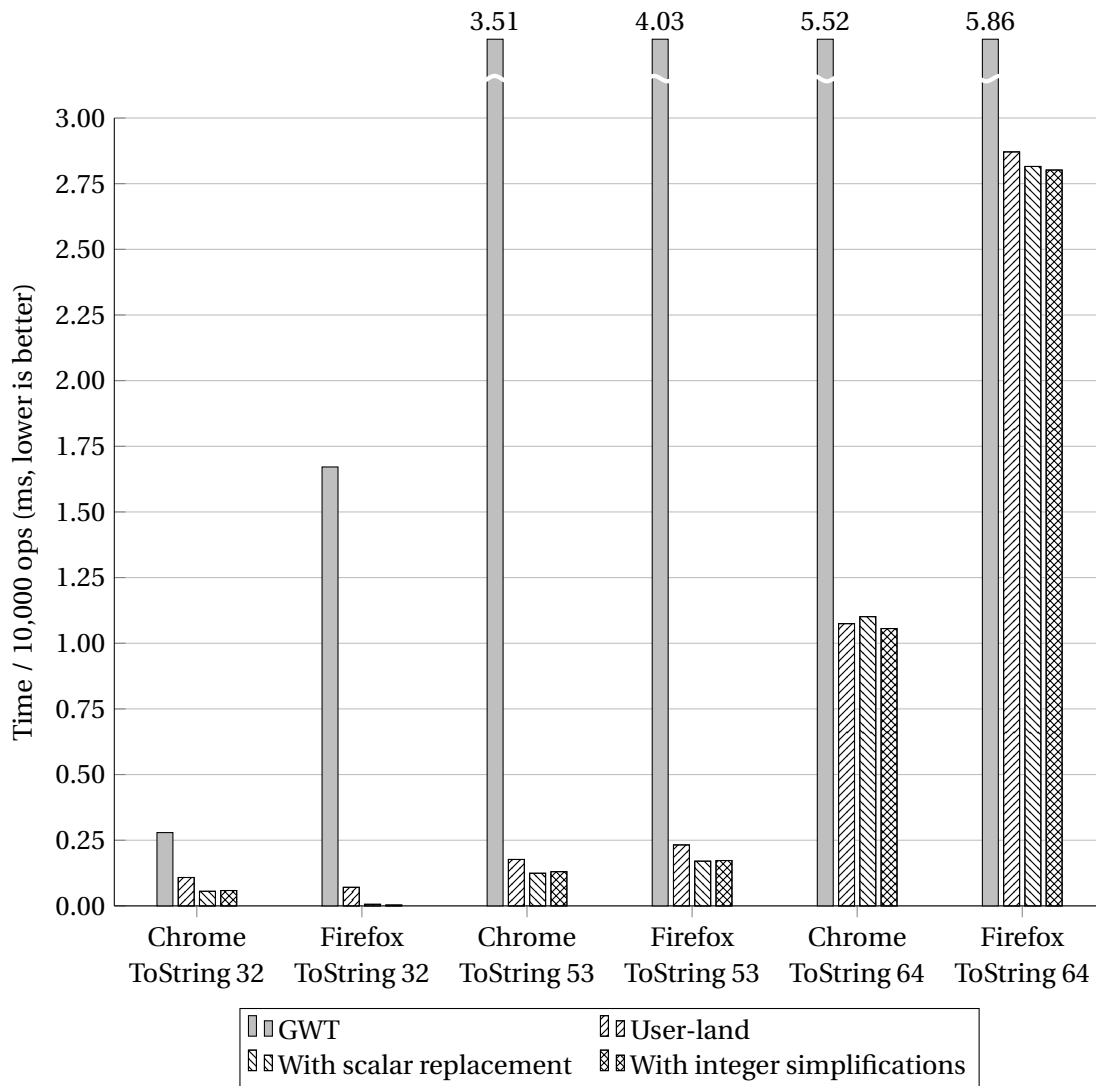
Figure 4.11 – Micro-benchmarks of our improvements to `toString`

which contains 3 branches on the longest path. Using clever but otherwise mundane rewritings (which are detailed as comments in the source code), we simplify it down to

```
def <<(n: Int): RuntimeLong = {
  if ((n & 32) == 0)
    new RuntimeLong(lo << n, (lo >>> 1 >>> (31-n)) | (hi << n))
  else
    new RuntimeLong(0, lo << n)
}
```

which has only one branch (and does not need an explicit n & 63). The skeptical reader will wonder whether the resulting operation is still correct. In Section 4.4.4, we mechanically verify the correctness of the optimized implementation of <<.

Addition

Recall from Section 4.4.1 that GWT had a significantly shorter implementation, because it could let the carry flow across the intermediate additions. The slow path for addition coming from TeaVM is comparatively verbose. Textbook instruction sequences for multi-precision additions use the very convenient adc instruction (add with carry). What if we could do the same? An optimal algorithm would look like

```
def +(b: RuntimeLong): RuntimeLong = {
  val lo = this.lo + b.lo
  val hi = adc(this.hi, b.hi)
  new RuntimeLong(lo, hi)
}
```

Essentially, adc amounts to:

```
val hi =
  if (overflow_occurred_in_lo_+)
    this.hi + b.hi + 1
  else
    this.hi + b.hi
```

We can detect whether an overflow occurred by testing if lo < this.lo, interpreted as an *unsigned* comparison. This is easily done by flipping the sign bit of the operands to <.

```
def +(b: RuntimeLong): RuntimeLong = {
  val lo = this.lo + b.lo
  val hi =
    if (lo ^ 0x80000000 < this.lo ^ 0x80000000)
      this.hi + b.hi + 1
    else
      this.hi + b.hi
  new RuntimeLong(lo, hi)
}
```

The resulting implementation contains less operations than the fast-path of TeaVM, making

the latter irrelevant.

As shown in Figure 4.9, this new implementation yields a 5x improvement to the addition on Chrome, bringing it on par with Firefox (for which it seems to have a slightly negative effect).

Multiplication

Both TeaVM and Kotlin have a multiplication algorithm that decomposes the product into 16-bit components. They do so in order to avoid the loss of bits due to wrapping around 2^{32}. Both algorithms require a total of 10 primitive multiplications, and a fair amount of additions and bitwise operations.

We can improve on this by decomposing some parts of the product into 32-bit components instead. We do lose some bits in the process, but this allows to accumulate the results of three 16-bit multiplications with a single 32-bit multiplication, if we can make use of the result. The derivation of our multiplication algorithm is unfortunately too long to fit here—it spans 200 lines of comments in the source code of `RuntimeLong`—but we include the final result, which contains only 6 primitive multiplications:

```scala
def *(b: RuntimeLong): RuntimeLong = {
  val alo = this.lo
  val blo = b.lo
  val a0 = alo & 0xffff
  val a1 = alo >>> 16
  val b0 = blo & 0xffff
  val b1 = blo >>> 16

  val a0b0 = a0 * b0
  val a1b0 = a1 * b0
  val a0b1 = a0 * b1
  val lo = a0b0 + ((a1b0 + a0b1) << 16)
  val c1part = (a0b0 >>> 16) + a0b1
  val hi = {
    alo*b.hi + a.hi*blo + a1 * b1 +
    (c1part >>> 16) + (((c1part & 0xffff) + a1b0) >>> 16)
  }
  new RuntimeLong(lo, hi)
}
```

To the best of our knowledge, this algorithm and its derivation are novel. Figure 4.9 shows that it brings a 4x speedup on Chrome and 2.3x on Firefox.

Division and remainder

The core of the division algorithm of GWT is the good old restoring division algorithm [68], adapted to compare before subtraction. This algorithm only works for unsigned divisions,

which is why GWT first needs to normalize the operands to be positive. Extra care must be taken for operands equal to `Long.MinValue`, which does not have an opposite in 2's complement.

We can however improve on GWT's algorithm with two major changes: first, we get rid of the `MinValue` special case by interpreting the 64 bits as unsigned after normalization; second, we short-circuit the loop as soon as the partial remainder fits in 53 bits.

For the first part, note that the bit pattern of `-MinValue`, interpreted as unsigned, is the correct mathematical value for the opposite of `MinValue`, i.e., 2^{63}. This means that, if the core division loop treats the 64 bits as unsigned, there is no need for any special case for `MinValue`, which streamlines the implementation and reduces code size. Doing so requires only one adaptation to the main loop: it needs to perform an *unsigned* `>=` rather than a signed one. Similarly to what we did for the addition, we can flip the sign bit and perform a signed comparison to achieve that result.

For the second part, we can easily by-pass the loop entirely when the dividend fits in 32 or 53 bits, as we can reuse primitive int or double division. TeaVM does this, which reduces the cases where the loop is needed to large values of the dividend. The major improvement that we contribute is to take advantage of the primitive double division even for dividends larger than 2^{53}. Let us look at the core loop and its invariant:

```scala
def divModHelper(a: RuntimeLong, b: RuntimeLong,
    isDivide: Boolean): RuntimeLong = {
  var shift = numberOfLeadingZeros(b) - numberOfLeadingZeros(a)
  var bShift = b << shift
  var r = a
  var q = new RuntimeLong(0, 0)

  /* Invariants:
   *   q >= 0
   *   If shift >= 0:
   *     bShift == b << shift == b * 2^shift
   *     0 <= r < 2 * bShift
   *   Else:
   *     0 <= r < b
   *   q * b + r == a
   */
  while (shift >= 0 && r != 0) {
    if (unsigned_>=(r, bShift)) {
      r -= bShift
      q |= (1L << shift)
    }
    shift -= 1
    bShift >>>= 1
  }
  if (isDivide) q
  else r
}
```

The idea of this loop is that it maintains the invariant $q \cdot b + r = a$. Eventually, we want q to hold the quotient `a/b`, and r to hold the remainder `a%b` (interpreted as unsigned integer operations). This is true if $0 \le r < b$, which is indeed the case after the loop, because either $r = 0$, or `shift` < 0, which implies $0 \le r < b$.

We can instead short-cut the loop as soon as $r < 2^{53}$ (what we call an "unsigned safe double", because we can safely convert it to a double without loss of precision). Note that this happens after at most 11 iterations of the loop instead of 64, a worthwhile improvement. Once $r < 2^{53}$, we can finish off the algorithm with a double operation:

```
while (shift >= 0 && !isUnsignedSafeDouble(r)) {
  ...
}
if (unsigned_>=(r, b)) {
  val rDbl = asUnsignedSafeDouble(r)
  val bDbl = asUnsignedSafeDouble(b)
  if (isDivide)
    q + fromUnsignedSafeDouble(rDbl / bDbl)
  else
    fromUnsignedSafeDouble(rDbl % bDbl)
} else {
  if (isDivide) q
  else r
}
```

If $r < b$, then we already have $0 \le r < b$ and nothing needs to be done. Otherwise, since $r < 2^{53}$, so is b, and we can perform the division on doubles, computing $q' = r/b$ and $r' = r\%b$. Since we know that $q' \cdot b + r' = r$ (by definition of unsigned `/` and `%`), and from the invariant, we have

$$q \cdot b + (q' \cdot b + r') = a$$

which can be rearranged as

$$(q + q') \cdot b + r' = a$$

Observe that this is the shape of the definition of quotient and remainder, with the quotient being $(q + q') = (q + r/b)$ and the remainder being $r' = r\%b$. This is what we return from the algorithm.

Finally, as a last optimization, we manually replace all the intermediate `RuntimeLongs` by their `Int` components, to reduce allocations and heap accesses.

Figure 4.10 compares the performance of our resulting division to GWT's original division (recall that our "best of breed" implementation was consistently 3x slower than GWT). Altogether, these optimizations bring up to an order of magnitude improvement.

Conversion to string

The last user-land implementation that deserves some remarks is the conversion to string in base 10. We have already mentioned that GWT processes the number in chunks of 10^9, leveraging the primitive conversion from `Int` to string. We can do even better by also leveraging the conversion of `Double` to string. Once we have normalized the input to a positive value a, if $a < 2^{53}$, we immediately short-cut to a double-to-string conversion. If $a \geq 2^{53}$, we perform *exactly one* long `divMod` operation by 10^9 to obtain $q = a/10^9$ and $r = a\%10^9$. Since we know that $2^{53} \leq a < 2^{64}$, we have that $2^{53}/10^9 \leq q \leq 2^{64}/10^9$, which implies that $0 < q < 2^{53}$. This means that q necessarily fits in a double, and r in an int. We can therefore delegate to two primitive conversions to string and a concatenation to conclude.

Figure 4.11 shows that this optimization brings a 5x improvement to the implementation of `toString` in the worst case, and up to 20x for the best case.

This concludes our tour of the various algorithms we have developed for the user-land implementation of `RuntimeLong`. Overall, improvements from 3x to 20x can be observed, compared to the best existing implementations. This improved implementation of `RuntimeLong` can be readily adopted by any language implementing `Long`s in JavaScript, in particular GWT, TeaVM and Kotlin.

4.4.3 Using an optimizing compiler

In the previous section, we have developed a user-land implementation of `Long`s targeted for JavaScript, which can be readily reused by other languages compiling to JavaScript. If we put some more restrictions on the language and its implementation, namely that it is statically typed and features an optimizing compiler, we can go further. To the best of our knowledge, Scala.js is the first implementation to apply an optimizing compiler to its user-land implementation of `Long`s. It would be possible for GWT, TeaVM and/or Kotlin to follow suit if they have a powerful enough optimizing compiler.

We apply optimizations in two phases, to show their relative benefits. In addition to the micro-benchmarks shown in Figures 4.9, 4.10 and 4.11, Figure 4.12 shows the results of a macro-benchmark: an implementation of SHA-512, straightforwardly ported from the C implementation in mbed TLS [43]. This hashing function makes heavy use of `Long`s, with a core loop mainly consisting of bitwise operations, shifts and additions. It is one of the few well-known algorithms requring the exact arithmetic modulo 2^{64} for `Long`s. The figure also shows benchmarks for "SHA-512-Int", which is a clone of "SHA-512" where every `Long` has been summarily replaced by an `Int`. The resulting algorithm is obviously wrong—it does not compute a correct SHA-512 hash anymore—but serves as an indication of the overhead imposed by `Long` operations relative to the primitive `Int`s.
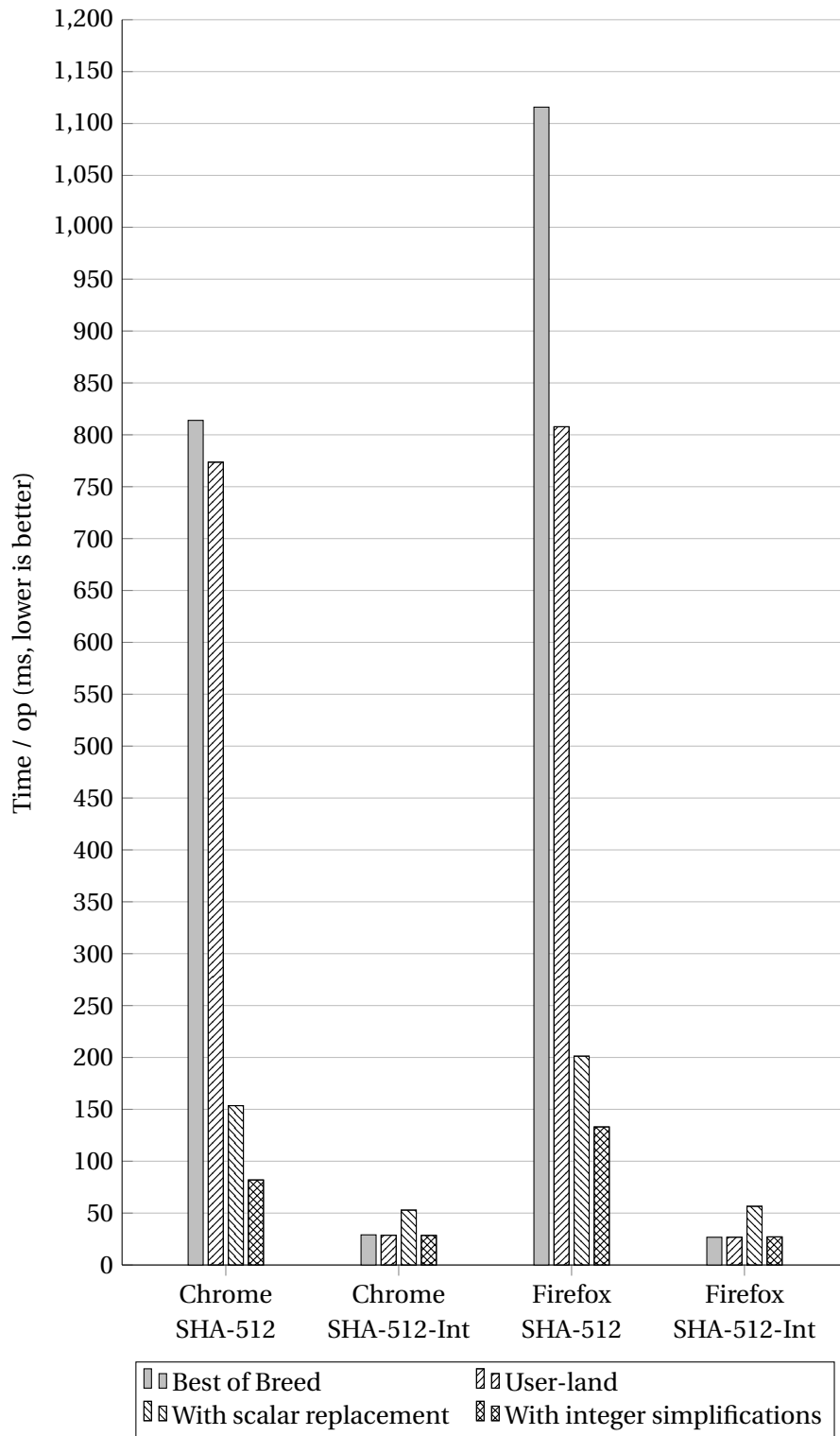
Figure 4.12 – The SHA-512 macro-benchmark on the various refinements of Section 4.4.2 and 4.4.3

First, we apply scalar replacement (also known as object inlining or stack allocation) to all local `RuntimeLong` values. For scalar replacement to be useful, we also need to inline the methods of `RuntimeLong`. The implementations of `/`, `%` and `toString` are much too large to be inlined, but all the other operations can be considered for inlining. Scalar replacement is a major optimization, well-known for its dramatic effects on performance. Not only does it avoid allocations and heap accesses, it also puts less pressure on the garbage collector.

Although our implementation of scalar replacement is generic and applies to other classes besides `RuntimeLong`, the latter is special-cased in the optimizer for additional effect, adding a little twist to this otherwise standard optimization. In general, when a reference escapes to another method, an optimizer has to backtrack and cancel the scalar replacement of the escaping value. This is necessary to preserve mutability and object identity. However, we know that `RuntimeLong` is immutable, and that its identity is never observable (since it is only an artifact of compiling primitive `Long`s). This means that, when a `RuntimeLong` escapes, we can create a new instance on the spot rather than backtracking. Likewise, as soon as we fetch a `RuntimeLong` from an unknown scope and store it in a local variable, we force its stack allocation as two primitive `Int`s. Figure 4.12 shows that scalar replacement of `RuntimeLong` brings a 5x improvement on Chrome, and 4x on Firefox.

In a second step, we add rewrite rules to simplify operations on `Int`s. This includes rules as simple and generic as `x & 0 == 0`, up to rewrite rules designed specifically after the code of `RuntimeLong`, such as `(x & 0xffff) >>> 16 == 0` (a pattern we find when multiplying by a small constant). We would expect JavaScript VMs to be able to perform this kind of optimizations on their own, yet our evaluation shows that this still brings an additional 2x speedup on top of the scalar replacement, both on Chrome and Firefox.

If we compare SHA-512 versus SHA-512-Int, we can see that `Long`s are overall 4.9x slower than `Int`s on Firefox, and only 2.9x slower on Chrome. This a very important result, as it shows that Scala.js' `Long`s are not prohibitively slower than `Int`s anymore. We can therefore have our cake and eat it too: correct 64-bit semantics and decent performance.

To put things in perspective, Figure 4.13 also compares the final implementation in Scala.js–with all the optimizations–against GWT, TeaVM and Kotlin. Scala.js is around 60x faster than GWT both on Chrome and Firefox. On Chrome, it is 8x faster than TeaVM, and 3.6x faster on Firefox.

4.4.4 Correctness

In Section 4.4.2 we outlined the various manual optimizations that shaped our user-land implementation. While these optimizations improve performance and code size, they also come at the cost of readability and result in considerably more complex code. To gain confidence that our implementation is correct, we mechanically verified most operations of `RuntimeLong` and provided a test suite for the rest.
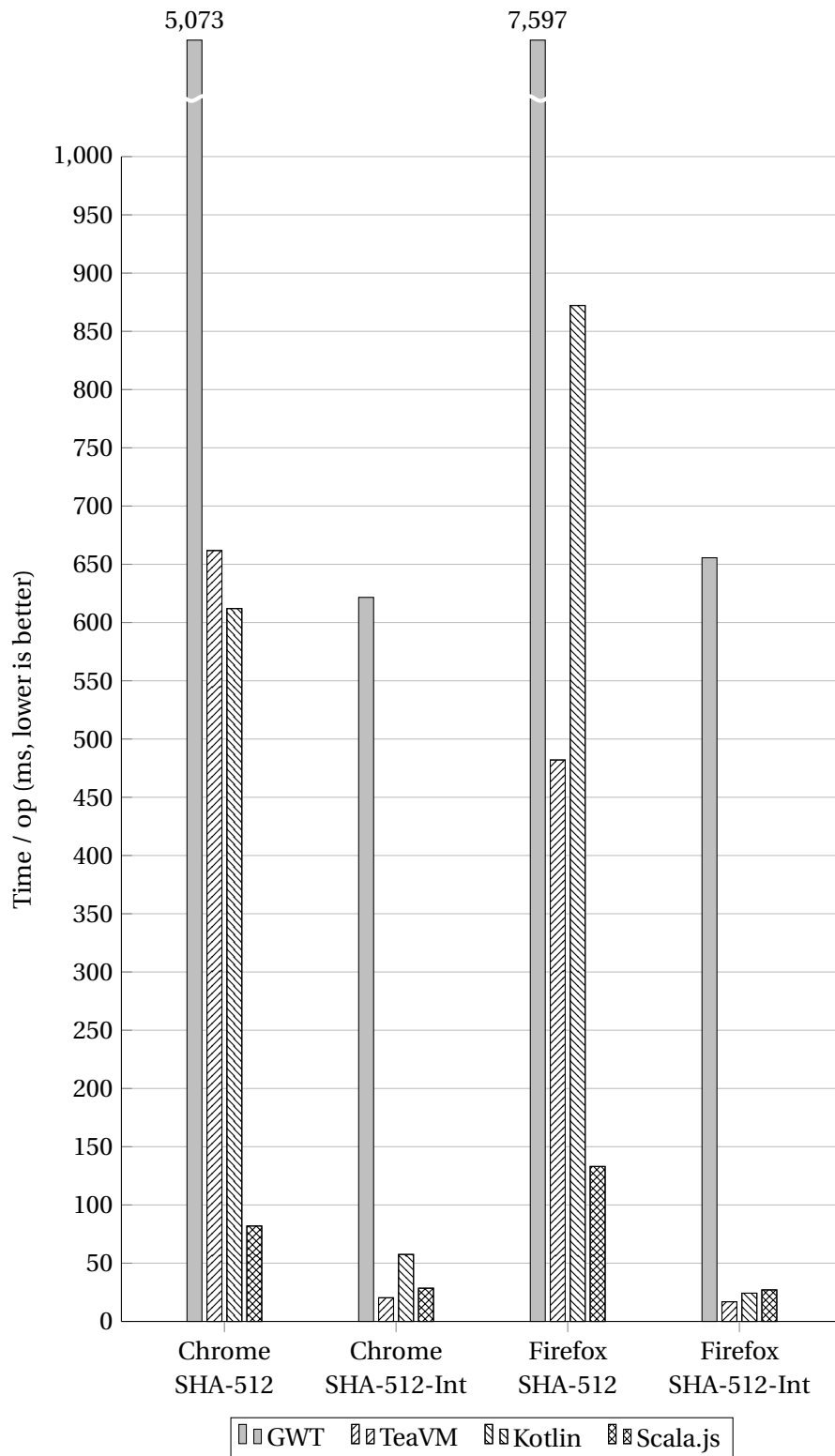
Figure 4.13 – The SHA-512 macro-benchmark on existing implementations–GWT, TeaVM and Kotlin–and the final Scala.js implementation with optimizations

Mechanical verification

Using Predicate-Qualified Types [56], a variant of refinement types for Scala, we proved the correctness of comparators, bitwise operators and arithmetic operators with the exception of `*`, `/` and `%`. That is to say, we verified that `RuntimeLong` faithfully implements the JVM's semantics of `Long`. To do so we extracted the relevant parts of the Scala.js implementation and added more precise type signatures which ensure the correspondence of semantics. The full specification—available in Appendix B—was then run through a Scala compiler supporting Predicate-Qualified Types.[9] Below we give a brief overview of how we used Predicate-Qualified Types for this verification task.

The purpose of `RuntimeLong` is to provide a drop-in replacement for Scala's `Long` type, which in turn corresponds to the JVM's primitive `long` type. We first define a conversion from `RuntimeLong` to `Long`:

```
val toLong: Long =
  (this.lo.toLong & 0xffffffffL) | (this.hi.toLong << 32L)
```

Using this conversion, we can straightforwardly specify correct behavior of an operation in `RuntimeLong` by requiring it to return the same result as the corresponding operation on `Long` when given the same inputs. For instance, in the case of the left-shift operator seen in Section 4.4.2, we write:

```
def <<(n: Int): {v: RuntimeLong =>
    v.toLong == (this.toLong << n.toLong)} = {
  if ((n & 32) == 0)
    new RuntimeLong(lo << n, (lo >>> 1 >>> (31-n)) | (hi << n))
  else
    new RuntimeLong(0, lo << n)
}
```

Note that we extracted the body of `<<` from our user-land implementation and added only a qualified (result) type to the method:

```
{v: RuntimeLong =>
  v.toLong == (this.toLong << n.toLong)}
```

The *qualifier* `v.toLong == ...` constrains what values belong to the type and hence may be returned. In our specific case it prescribes that performing a left-shift by n bits using the implementation in `RuntimeLong` and converting the result to a `Long` (`v.toLong`) must give the equivalent result as converting the original `RuntimeLong` to `Long` first and performing the left-shift only then (`this.toLong << n.toLong`).

The compiler will then try to prove that the implementation adheres to this specification by inferring precise qualified types for the possible return values in both branches of `<<`. The

---

[9]The current version of the Predicate-Qualified Type system is available at https://github.com/gsps/dotty/tree/liquidtyper.

resulting subtyping checks are then translated to validity queries in a logic over bitvectors and uninterpreted functions. The system discharges such proof obligations using SMT solvers such as CVC4 [4] or Z3 [12].

Limitations

Unfortunately we have not yet succeeded in verifying the implementation of all the operations of `RuntimeLong`. Two obstacles for a fully mechanical proof remain: currently the Predicate-Qualified Type system translates the multiplication and division methods to SMT queries that neither CVC4 nor Z3 can solve in a reasonable amount of time. We hope to remedy this in the future by providing adequate hints to the solver. Moreover, our fine-tuned implementation of division relies on conversions to and from `Doubles`, which further complicate the SMT queries.

Test suite

To mitigate these limitations we also built a test suite comprised of hundreds of randomly generated test cases. The expected results of operations are computed on a JVM, then stored as unit tests in the Scala.js test suite. This is essentially a manual and static variant of differential testing [44]. The test suite also includes manually constructed tests for known corner cases of `RuntimeLong`'s implementation (typically around overflow cases).

In addition to filling the gaps left by our verification tool on multiplication, division and `toString`, the end-to-end test suite gives confidence about the optimizer's rewritings described in Section 4.4.3.

Finally, the `Longs` are implicitly tested by the larger test suite of the entire Scala.js platform, which includes an implementation of big numbers relying on `Longs`.

4.4.5   Related Work

The art of compiling multi-precision arithmetic data types on top of single-precision ones is decades-old [31, 68]. However, all well-known algorithms assume that the primitive operations for single-precision arithmetics offer some support to build multi-precision operations on top of them. For example, they assume that addition sets a carry flag that can be recovered without branching (e.g., via `adc` on x86); or that multiplication outputs two single-precision registers for the lo and hi parts of the results. When compiling to a higher-level language such as JavaScript, the target does not offer such facilities, and we need to compute the carry ourselves.

One area we could explore, but as of yet have not, is the ahead-of-time optimization of

divisions by a constant, using techniques ranging from using simple shifts to multiplication by the inverse [28]. The latter technique is not directly applicable, though, as it also assumes that multiplication gives access to the high half of the result. It might be worth implementing this multiplication in software, though, if it means avoiding the 10x slowdown of division over other arithmetic operations.

On a different note, compilation to JavaScript is commonplace, nowadays. When it comes to compiling down 64-bit integers from the source language, there have been two different strategies. One is to emulate them in software, in the tradition of GWT [25]. When GWT was first released, Longs were already known to be extremely slow, and GWT's documentation warns against using them too liberally. This strategy has been followed by surprisingly few implementations, e.g., TeaVM [62] and Kotlin [33]. On the more academic side, we find two other compilers doing so: Doppio by Vilk and Berger [66] and the Graal AOT JS compiler by Leopoldseder et al. [36]. Both papers mention Longs as a significant performance bottleneck. Emscripten [72] initially had incorrect 64-bit integers, but they later added compilation modes to choose between fast-but-incorrect and correct-but-slow. Remarkably, industrial and academic compilers alike seem to accept the terrible performance of Longs as a fact of life. In Scala.js, we were never satisfied with the performance of our Longs, and have always tried to improve them, eventually achieving the results presented here. After we pointed out our implementation to Doppio's authors, they decided to look into adopting it.[10]

The other and sadly more popular strategy adopted by implementers has been to disregard correctness. Instead, Longs are typically compiled down to JavaScript numbers, i.e., Doubles, effectively dropping the precision down to 53 bits. Even languages specifically designed to cross-compile to JavaScript have been making that trade-off, such as Ceylon [54]. This has been justified by the fact that correctly compiling 64-bit integers yields code that is much too slow. With an implementation such as the one developed in this section, this should not be a concern anymore, obviating the need for such a compromise on correctness.

A radical alternative would be to compile to WebAssembly [29] instead of JavaScript. This seems like a no-brainer, since WebAssembly features native 64-bit integers. However, it does not (yet) have any support for garbage-collected languages, nor for interoperating with arbitrary JavaScript values, which are features that Scala.js, GWT, TeaVM and Kotlin all need.

4.4.6 Conclusion

We have presented a fast implementation of 64-bit integers, aka Longs, for the JavaScript platform, which we use in the Scala.js compiler and optimizer. The implementation relies on two key components: an efficient user-land implementation—derived as refinements of the state-of-the-art implementations—, and an optimizing compiler capable of inlining and scalar replacement at the least. The implementation is 3.6x to 60x faster than existing approaches,

---

[10]https://github.com/plasma-umass/doppio/issues/444

and, crucially, is only about 3–5x slower than native 32-bit integers.

To ensure correctness of our solution, we have mechanically verified parts of the user-land implementation using Predicate-Qualified Types for Scala. The mechanical proofs are complemented by a test suite based on differential testing.

Our fast implementation of Longs solves the years-old tension between correctness and performance for compilers targeting JavaScript. At last, we can have our cake and eat it too: the correct behavior of wrapping 64-bit integers, and the performance of the resulting code.

4.5   Performance results

We conclude this chapter about compiling the IR down to JavaScript with a performance analysis of the produced JavaScript programs. The measurements are done using a series of benchmarks:[11]

- 3 benchmarks ported from the Octane suite [47]: DeltaBlue, Richards and Tracer.
- the SHA-512 and SHA-512-Int benchmarks, which we already used in Section 4.4.3.
- a number of benchmarks from Cross-language Compiler Benchmarking: Are We Fast Yet? by Smarr et al. [41], ported to Scala by Denys Shabalin and Olivier Blanvillain.

The danger with benchmarks, when they drive optimizations, is to fall into the trap of Goodhart's law, i.e., overfitting optimizations to improve results on particular benchmarks, although said optimizations have little to no effect on other actual codebases. In order to mitigate this problem in this thesis, we use new benchmarks for the analysis, that were never measured before, in addition to benchmarks that have helped us analyze optimization opportunities in the past. The historic benchmarks are DeltaBlue, Richards, Tracer, SHA-512 and SHA-512-Int. The new benchmarks are all the other ones, taken from Smarr's benchmark suite.

Each benchmark is cross-compiled with Scala/JVM and Scala.js, so that we can use Scala/JVM as a baseline. In addition, the benchmarks coming from Octane are compared to their original JavaScript implementation, as another baseline. In all cases, we use Scala 2.12.5. For the Scala.js version, we apply a mix of different configurations that affect performance:

- The ECMAScript 5.1 output versus the ECMAScript 2015 one.
- Development mode versus production mode (i.e., checks for undefined behavior on/off).
- The Scala.js optimizer on/off.
- The additional Google Closure Compiler (GCC) on/off.

All JavaScript artifacts are run in three different environments: Node.js v10.0.0 (using V8 v6.6.346.24), Chrome v65.0.3325.181 (using V8 v6.5.254.3), Firefox v59.0.2. JVM measurements

---

[11]Their source can be found at https://github.com/sjrd/scalajs-benchmarks.

were done using Oracle® Java HotSpot™ 64-Bit Server VM v1.8.0_40. In all cases, measurements are taken on an Intel® Core™ i7-4790 CPU @ 3.60GHz with 32 GB of RAM, running Ubuntu 16.04.4 LTS. Benchmarks were run multiple times, with a warmup period and a measurement period, so that the standard error of the mean (SEM) of the measurement period was of the order of 1% or less of the absolute values. We then normalized the mean running times of JavaScript artifacts with respect to the measures on the JVM. This means that a normalized running time > 1 is slower than the JVM and < 1 is faster than the JVM.

4.5.1 Effects of performance configurations on various platforms

To kick off discussions of our results, we look at the effect of the performance configurations of Scala.js on various platforms. For reasons that will become clear in Section 4.5.2, we only use the ECMAScript 5.1 output of Scala.js here. More specifically, we analyze the following configurations:

- development mode, no optimizer, no GCC, which is supposed to be the slowest configuration;

- development mode, with optimizer, no GCC; this is the default development configuration, aka `fastOptJS`;

- production mode, no optimizer, no GCC;

- production mode, no optimizer, with GCC;

- production mode, with optimizer, without GCC;

- production mode, with optimizer and with GCC; this the default production configuration, aka `fullOptJS`.

The pattern-code is that the bar is filled with gray ▊ for development mode or white ☐ for production mode; it contains north-east lines ▨ if it uses our optimizer; and it contains north-west lines ▧ if it uses GCC. The patterns combine to give the full pattern of a configuration.

We do not include measures in development mode with GCC because they are irrelevant from a practical point of view, and do not give any interesting insight either.

On Node.js

Figures 4.14 and 4.15 show the results on Node.js. They are separated in two because of the differing scales.

The first thing that stands out is the mandelbrot benchmark. It has a surprising ability to ignore any kind of optimization that we apply to it, and exhibits a completely flat profile. This can easily be explained by looking at what this benchmark does: it is contained in one
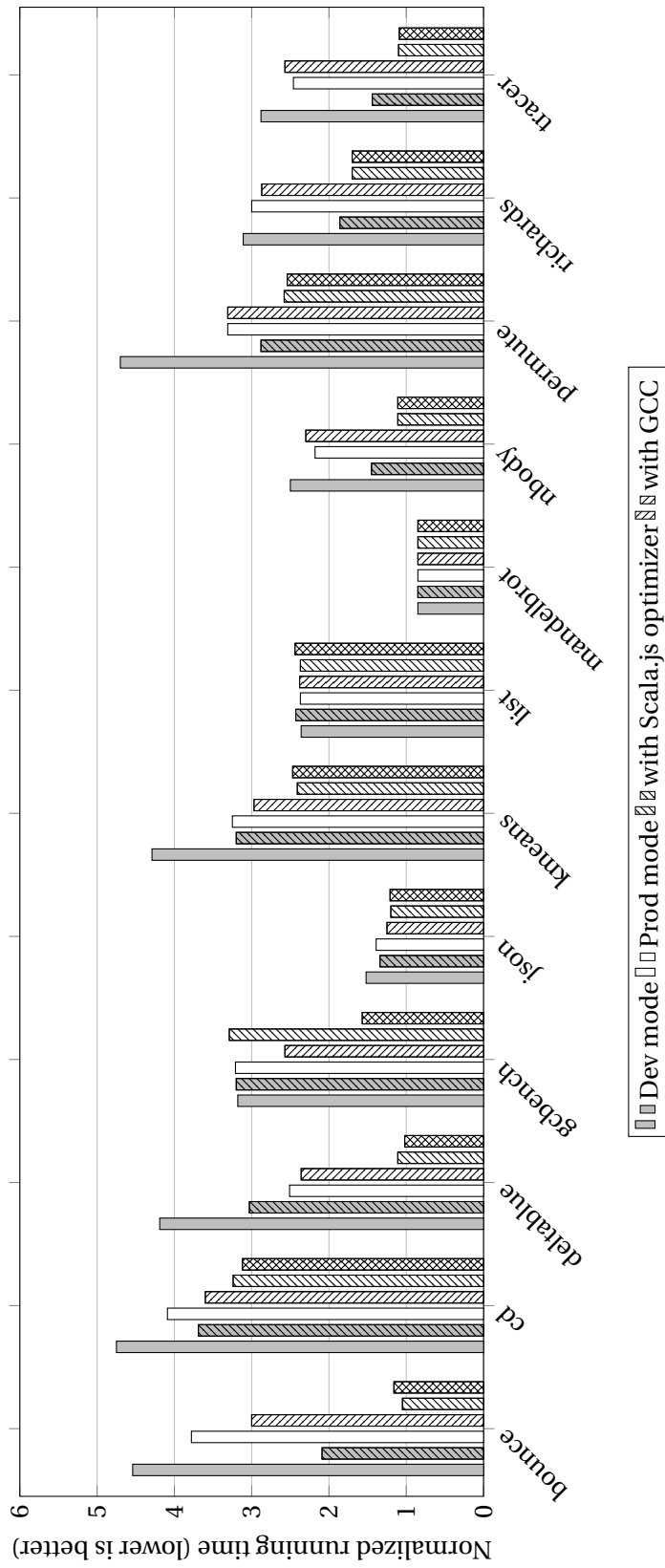
Figure 4.14 – The effects of several optimization options on Node.js (part 1: the fast ones)
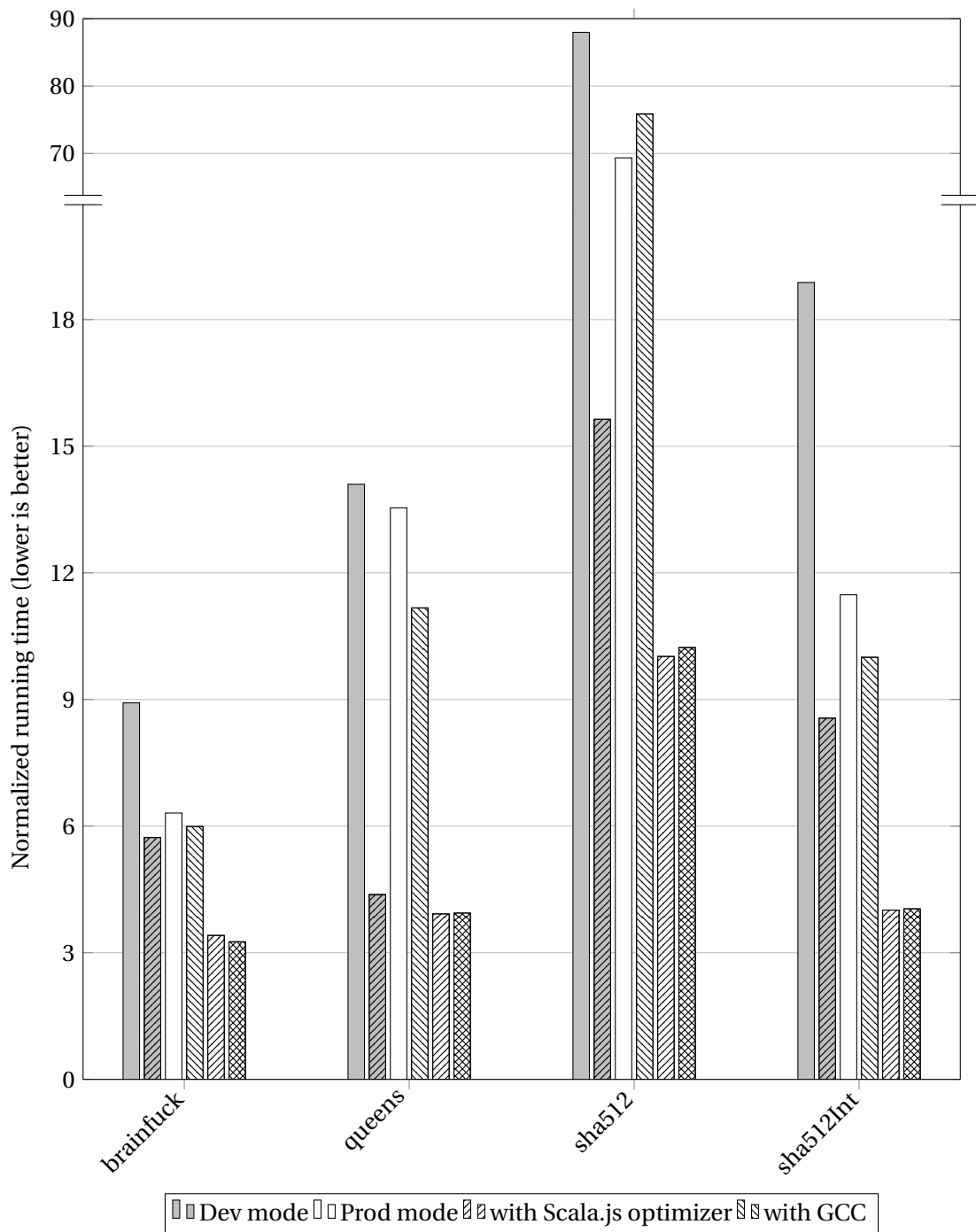
Figure 4.15 – The effects of several optimization options on Node.js (part 2: the slow ones)

method, and performs `while` loops with operations on local variables of type `Int` and `Double`. Any optimizer looking at that code would say "Aye, good job, nothing to see here," and move on. Moreover, it does not perform any operation that can trigger undefined behavior, so there is no difference between development and production mode, either. Another surprising observation on mandelbrot is that the flat line is *under 1* (it lives at 0.85 to be precise) which means it is consistently faster than the JVM. It is difficult to imagine why the JVM does not perform equally well, but it might simply be that the JVM is not very good at `Double` operations, which make for the most part of the benchmark, compared to V8. This is plausible because `Double`s are the only primitive numbers in JavaScript, forcing the JS VMs to go to great lengths to optimize them, whereas a JVM would focus on integer arithmetics.

The other benchmark exhibiting a flat profile is list, however this one is around 2.4x slower than the JVM. Contrarily to mandelbrot, list is designed to perform as many heap operations as possible, in a way that cannot be stack-allocated without partially evaluating the entire program.

The flat benchmarks notwithstanding, one trend can be observed across all benchmarks: applying GCC without the Scala.js optimizer typically brings some improvements over the production baseline (▯ → ▨), but when the Scala.js optimizer is enabled, applying GCC makes no difference (▨ → ▨). In any case, applying the Scala.js optimizer brings substantial improvements. This is a compelling argument for the existence and usefulness of our optimizer, and clearly shows that the optimizations it can incrementally perform (and in parallel) have a real impact on the performance of resulting programs. The effects of our optimizer are particularly spectacular on benchmarks that are very slow without it (those shown in Figure 4.15). For example, queens and sha512Int would be 12–13x slower than the JVM without our optimizer, but are brought down to being "only" 4x slower with it. Of course, the most remarkable effect is that on sha512—note the break in the y axis along with the change of scale—, with a whopping 7x improvements (which, unfortunately, is only enough to get our performance to 10x slower than the JVM, proving that there is still work to do in that area).

There is one major exception to the previous observations: gcbench. For some reason, this benchmark performs very poorly without GCC (across all levels of optimizations of Scala.js itself). GCC without our optimizer is marginally better, but only the combined might of the Scala.js optimizer and GCC can bring it down to acceptable performance. The reason for this combined effect is unclear to us, but is consistently observed across Node.js, Firefox and Chrome, suggesting that it is a real effect.

We can also observe that the development mode receives significant benefits from the Scala.js optimizer (going from ▮ to ▨), which justifies its use in incremental runs during iterative development—which, in turns, justifies the efforts put into making it incremental in the first place! Moreover, we can see that applying the optimizer on dev mode ▨ is virtually always more beneficial than using the production mode without the optimizer ▯. This is also a compelling argument to use the incremental optimizer during development, rather than removing checks

that can be useful for debugging purposes.

Perhaps the most *disappointing* result is the cd benchmark. Despite the performance in production mode without optimizer ⬚ being only 4x slower than the JVM, the fully optimized version is still more than 3x slower. There seems to be a lot of opportunities there that our optimizer is not capable of leveraging, resulting in a program that is probably slower than expected. On the bright side, it means that we should be able to improve our optimizer by looking at what it is missing in that codebase.

On the other hand, the most satisfactory result is definitely the bounce benchmark. While being 4.5x and 3.8x slower than the JVM without optimizations (in development and production mode, respectively), applying our optimizations brings it down to 2x slower in development mode, and to a mere 5% slower in production mode. This shows that our optimizer has the power to bring the performance of Scala.js from "slow" to being on par with the JVM at least in some cases.

In browsers

Unsurprisingly, the results for all benchmarks are extremely similar between Node.js and Chrome, given that they both use V8, and versions of it not far apart from each other. They are so similar that we do not discuss them here, nor even show the graphs.

Figures 4.16 and 4.17 show the results on Firefox.

Compared to Node.js, one thing stands out: the numbers are larger, in other words, Firefox seems to be slower across the board on Scala.js benchmarks. This does not necessarily mean that Firefox is overall slower, as it could be that we overfitted the code emitter of Scala.js to V8. This is all the more plausible due to the fact that it is so much more convenient to run benchmarks through Node.js in automated ways, so we tend to measure V8 more often.

Across all benchmarks, we observe that going from development mode to production mode has less impact than on V8, whether without the optimizer (⬚ → ▨) or with it (⬚ → ▨). This suggests that Firefox' optimizer is mostly capable of identifying our checks for undefined behaviors as guards that are never actually triggered, and using that knowledge to perform more optimizations. One exception is sha512Int with the optimizer enabled, where enabling production mode brings the running time from more than 10x down to 3.5x (which, in this case, is faster than Node.js).

The most disappointing and more satisfactory benchmarks (cd and bounce, respectively) keep their corresponding crowns on Firefox. In fact, the results on bounce are even more spectacular than they were on Node.js, with a 6x improvement brought by our optimizer. mandelbrot, although slightly slower than on Node.js, is still a flat line under 1.
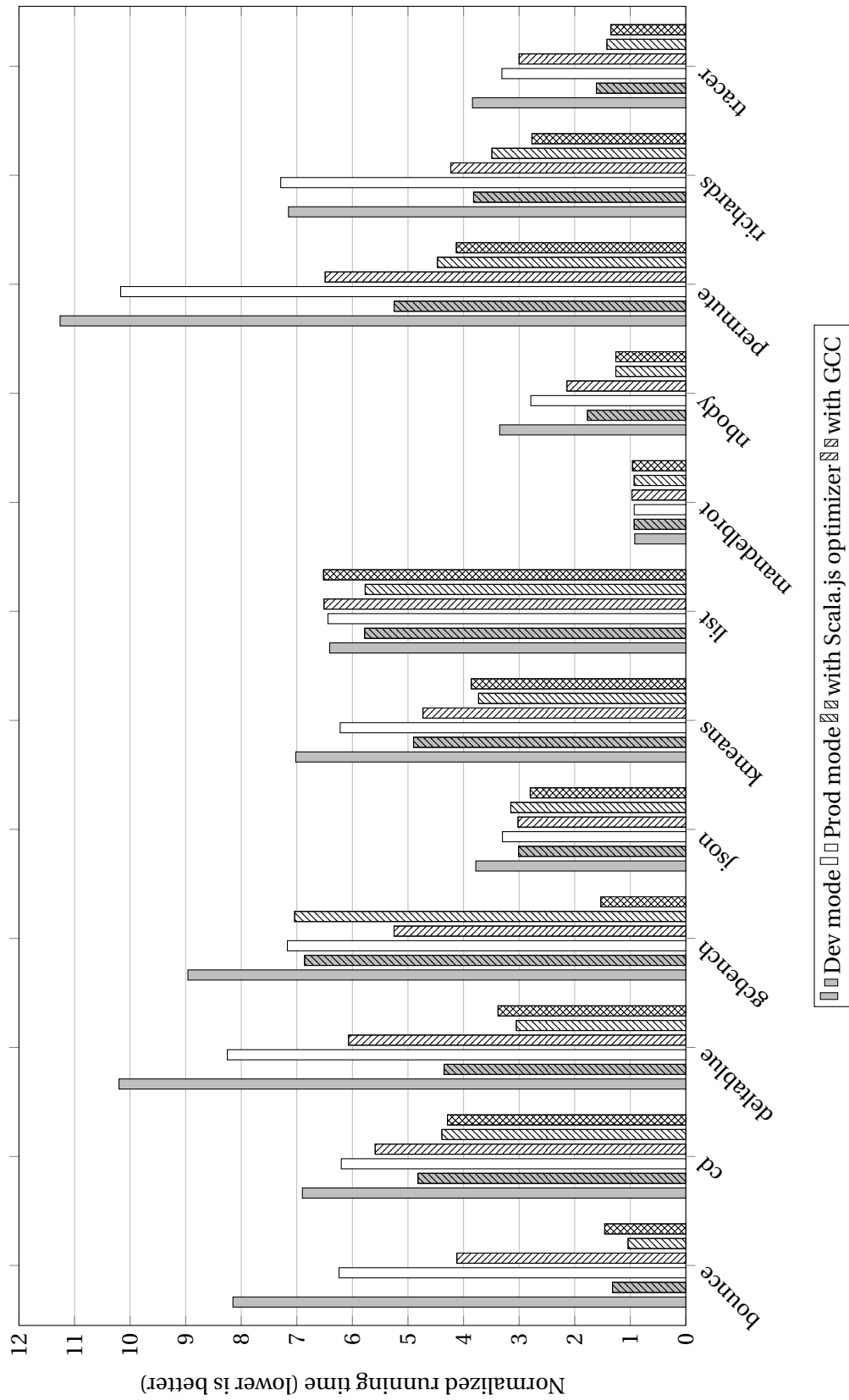
Figure 4.16 – The effects of several optimization options on Firefox (part 1: the fast ones)
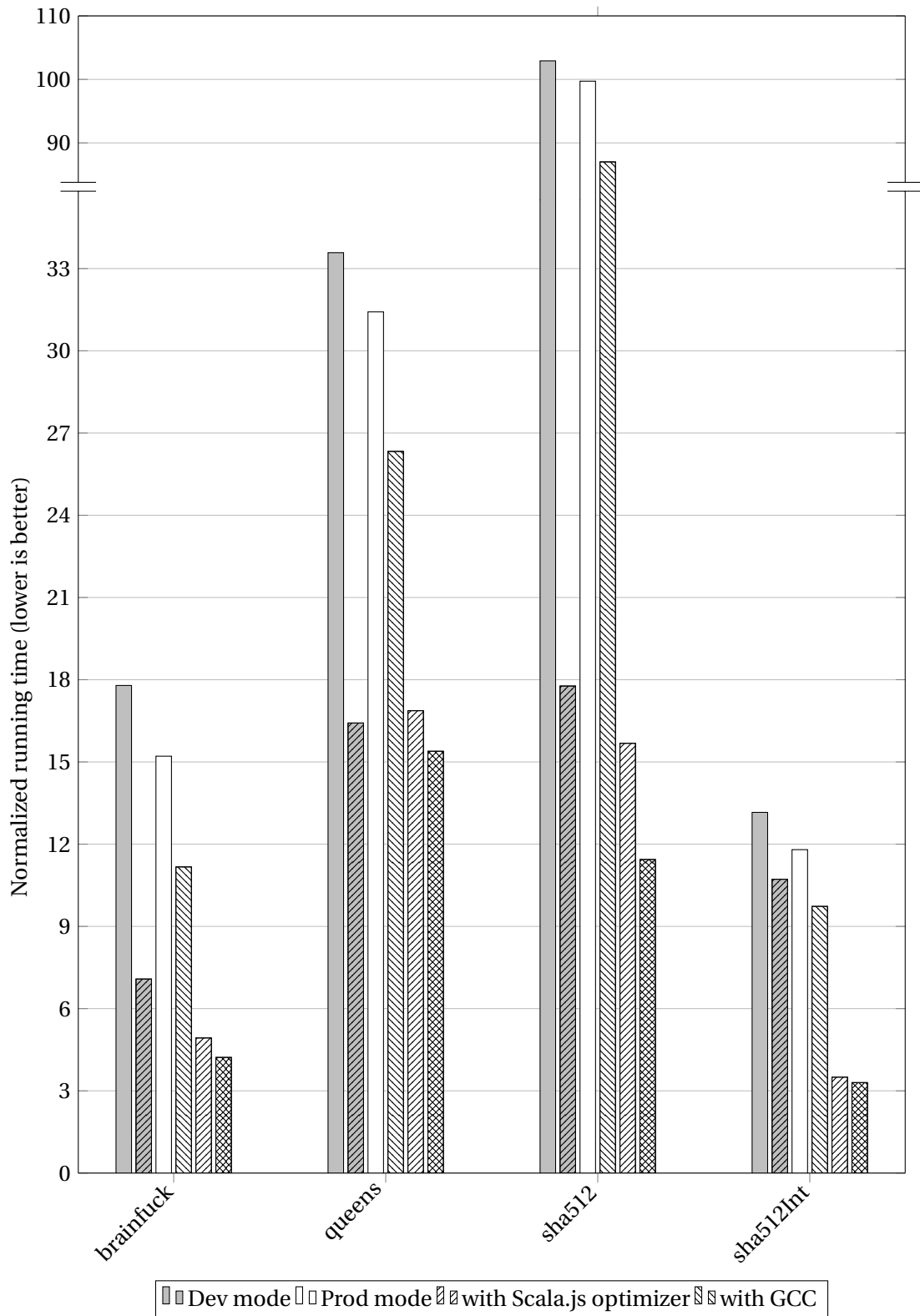
Figure 4.17 – The effects of several optimization options in Firefox (part 2: the slow ones)

4.5.2 ES 5.1 features versus ES 2015 across engines

Scala.js can be configured to emit JavaScript code that leverages ECMAScript 2015 features where it makes sense, or to stick to ES 5.1 features. By default, the ECMAScript 5.1 output is used, because, as we will see in this section, Firefox really does not like ES 2015 features yet. Since the Scala.js compiler cannot yet use GCC with its ECMAScript 2015 output, we only measure without GCC. For the purposes of studying ES 5.1 versus ES 2015 across engines, we stick to the configuration with production mode and Scala.js optimizer but without GCC (which was ▧ in the previous section).

Figure 4.18 shows the result. We depict ES 5.1 in gray ▯ and ES 2015 in white ▯, and we add north-east lines ▧ for Chrome and north-west lines ▨ for Firefox (leaving no-lines for Node.js). The y axis had to be truncated for readability, because of the dreadful performance of Firefox with ES 2015, which exhibits slowdowns of up to 35x compared to ES 5.1.

The general trend across all benchmarks is that running time increases as we go "from left to right", i.e., Firefox is slower than Chrome which is slower than Node.js, and the ES 2015 variant of each is slower than their respective ES 5.1 variant. However, for Node.js and Chrome, the performance loss of using ES 2015 is usually very small if not nonexistent. The bounce benchmark is even slightly faster with ES 2015 on both V8-based environments, as well as richards on Chrome.

The exceptions worth noting are:

- The queens benchmark poses a significant problem to Node.js and Chrome in its ES 2015 version, with slowdowns of more than 3.5x. The particularity of queens is that it uses so-called non-local `return`s in the Scala code, which are translated into `throw` and `try..catch`. This could explain the issue in ES 2015 if V8 has trouble optimizing `let`s and `const`s in the presence of `try..catch` and/or `throw`.

- Firefox likes the ES 2015 version of the list benchmark more than its ES 5.1 version. Nevertheless, it is still significantly slower than Node.js and Chrome.

- Once again, mandelbrot resists any attempt of being swayed in any direction.

4.5.3 Comparison with hand-written JS across engines

Finally, we compare the performance of the fastest configurations of Scala.js (ES 5.1, production mode with optimizer, with and without GCC) against the original hand-written JavaScript version. We limit this analysis to the 3 benchmarks ported from the Octane suite, since they are the only ones for which we have an original JavaScript to compare against. Given the similarities between Node.js and Chrome, and because there are only so many patterns we can fill the bars with, we omit Chrome in the graphs.
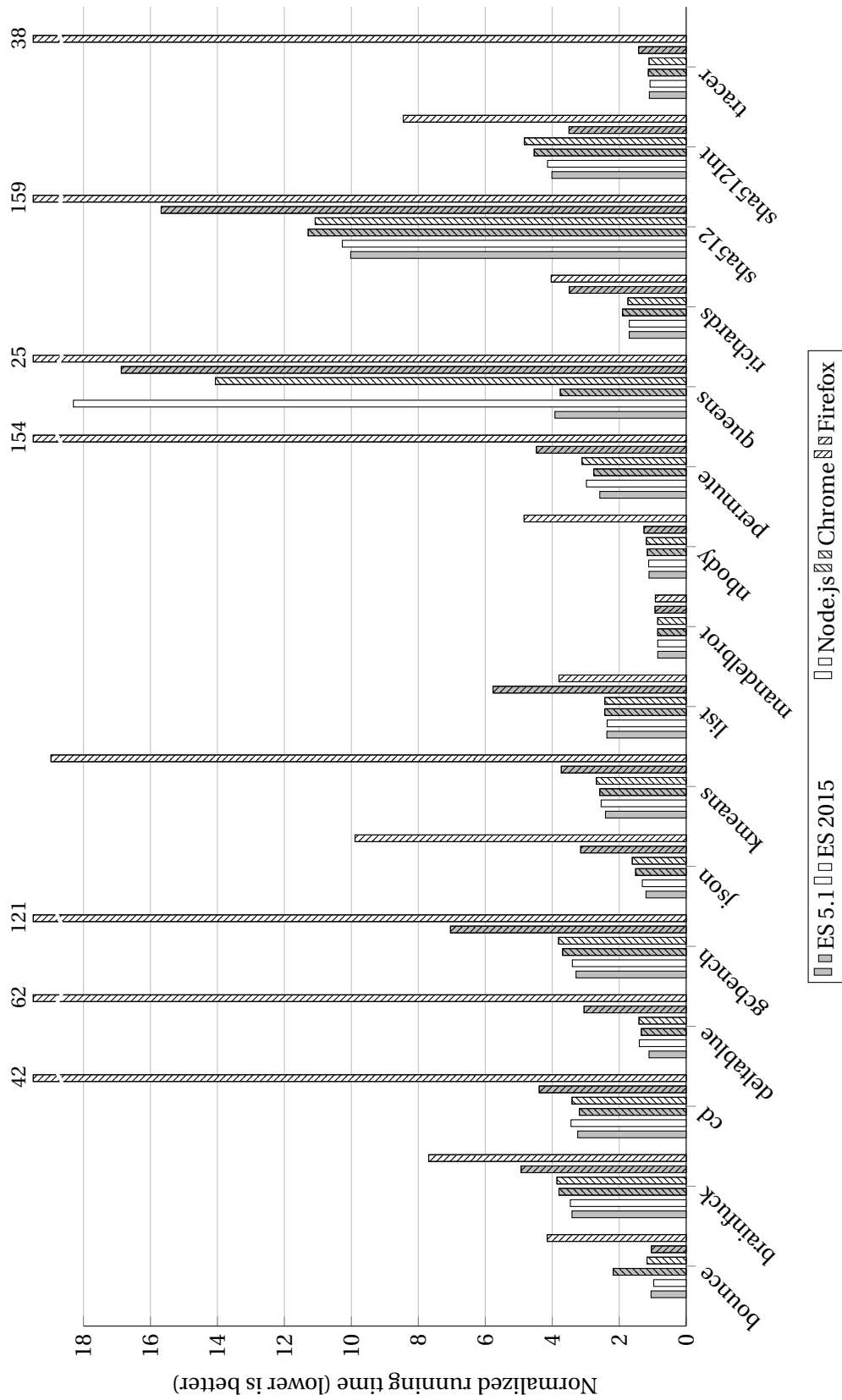
Figure 4.18 – ECMAScript 5.1 versus ECMAScript 2015 across engines
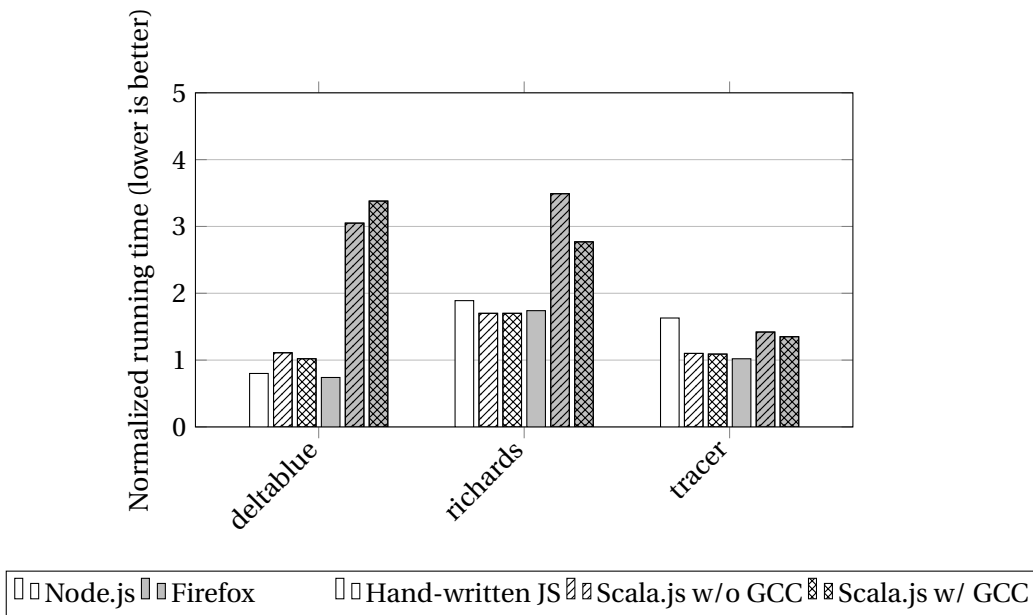
Figure 4.19 – Scala.js versus hand-written JavaScript on Node.js and Firefox

We represent Node.js in white ▢ and Firefox in gray ▮. We use no lines for hand-written JavaScript ▢, north-east lines for Scala.js without GCC ▨ and both north-east and north-west lines for Scala.js with GCC ▩. Figure 4.19 shows the results.

The results are quite remarkable on Node.js, with Scala.js being faster than hand-written JavaScript on two benchmarks, and 1.4x slower on the third one. The tracer benchmark is our favorite: Scala.js is 33% faster than the hand-written version. This is probably due to the numerous opportunities for scalar replacement of intermediate vectors and points in that benchmark.

On Firefox, the situation is reversed. Scala.js performs much worse than the hand-written JavaScript. Combined with the earlier observations that Scala.js was generally slower on Firefox than in V8-based environments, this strongly suggests that we overfitted our compiler for V8. Later developments should take more care about the other engines. It is unclear at this point what specific decisions are detrimental on Firefox.

Chapter 5

Conclusion

In this thesis, we have presented the design of Scala.js for portability with respect to Scala/JVM and interoperability with JavaScript, while keeping performance concerns in mind. While we have focused on the particular case of Scala.js, we are confident that the design methodology that we have followed would easily transfer to other languages targeting the JVM and JavaScript. More broadly, we hope that the most important insights can be transferred to any language targeting multiple platforms. The methodology involves two key aspects. First, we want our cross-platform language to be portable across runtimes, which means that most programs should cross-compile and behave the same way on all platforms. Second, we want the language to be interoperable with each platform it targets.

Sometimes, these two goals conflict, and some trade-offs must be made. If we trade some portability away to improve interoperability, we must validate that our choices are well-received by the ecosystem of libraries, in order not to threaten the portability of the entire ecosystem. Trading away interoperability is a bigger problem, as it can wholesale prevent from using some libraries of the host target. In that case, we must ensure that every semantics expressible in the target platform's language can be expressed in our language, even if some of it might be inconvenient or verbose, i.e., that the interoperability feature set is complete with respect to the host language. Overall, the language should be entirely interoperable, and as portable as possible.

To be able to optimize a language with deep interoperability features such as Scala.js, we designed an intermediate representation which gives first-class support both to portable, statically typed structures as well as interoperable, dynamically typed operations. An optimizer can reason about the entire language to perform sound optimizations even in the presence of interoperability features calling into the unknown world of the host language. This is in contrast with what we call FFI-based interoperability (Foreign Function Interface), where the FFI does not integrate well enough into the semantics of the intermediate representation, forcing an optimizer to bail on a lot of optimizations in the neighborhood. In the particular

case of Scala.js, we designed the language and IR to essentially live in a closed world, allowing advanced whole-program optimizations that stay sound in the presence of unknown host language code.

We evaluated the performance of programs compiled by Scala.js across various levels of optimizations, comparing against Scala/JVM and some hand-written JavaScript versions. Fully optimized Scala.js programs were found to be competitive on V8-based engines, being at most 4 times slower than the JVM (often less than 2.5x slower and up to 15% faster) and within the 0.7–1.4x range with respect to hand-written JavaScript. The exception being SHA-512, a benchmark for 64-bit integers, which, despite the breakthrough studied in Section 4.4, is still 10x slower than on the JVM. On Firefox, Scala.js programs were found to perform comparatively poorly, which suggests that our compiler has been overfitted to V8 and needs more work on other engines.

5.1 Perspectives

Scala.js has been quite successful in the industry. It has been well-received in the Scala ecosystem, and is supported by most of the major Scala libraries, thanks to its dedication to portability. Moreover, developers have been using Scala.js on all sorts of JavaScript-based runtimes, which is a testament to its interoperability: client-side in browsers, servers using Node.js, mobile apps through React Native, desktop apps with Electron, Chrome extensions, command-line applications, etc. The level of adoption and enthusiasm that Scala.js has received has surpassed what we could have hoped for, and that is likely the best reward of this thesis. It is also a great responsibility, as Scala.js has become a product that will outlive this thesis by many years. As such, the design and development will need to adapt to new versions of Scala and ECMAScript, which will bring new requirements in terms of portability and interoperability.

For each new version of Scala, we must adapt the Scala.js compiler front-end to support the new internal APIs and potentially new language features. We have already been doing this for five years, as we have evolved from Scala 2.10.2 at the inception of Scala.js to Scala 2.12.6 at the time of writing. Usually, this involves only mundane engineering work, although occasionally the changes have large impacts on Scala.js. The canonical example would be the transition from 2.11 to 2.12, which required to add support for default methods in the Scala.js IR. Because Scala.js has now acquired a solid place in the Scala ecosystem, evolutions to the Scala language, shepherded by the SIP Committee, now take Scala.js concerns into account. This ensures that new versions of Scala will not become incompatible with Scala.js, and that there is always a path forward. The upcoming Scala 3 will require a rewrite of the compiler plugin, which converts compiler trees into the IR, but should not fundamentally affect Scala.js.

New versions of ECMAScript are potentially more problematic, as its evolution is agnostic of Scala.js. We must ensure that Scala.js stays up-to-date, always providing new interoper-

ability features to cover the whole set of semantics expressible in ECMAScript source code. Fortunately, most ECMAScript language enhancements do not bring fundamentally new run-time semantics: syntactic sugar and other language constructs which can be compiled down to existing semantics do not require changes in Scala.js. An example of such a construct would be the new number literals of the Numeric Separators proposal.[1] Moreover, many fundamentally new semantics are made available as library functions and classes rather than language constructs. Those improvements are readily accessible to Scala.js, since it can talk to any JavaScript library, including any addition to the standard library. An important example here would be Weak References.[2] The only changes that we need to look out for are language changes with semantics that were previously not expressible. We have already an example of this category: `new.target`! Another proposal for the future of ECMAScript that falls into that category is `import.meta`.[3] For those changes, we will need to design corresponding interoperability features, spanning the whole pipeline: at the source level of Scala.js, in the Scala.js IR, and in the compiler back-end with its optimizer and emitter.

Whether Scala.js evolutions are driven by changes in Scala or ECMAScript, backwards binary compatibility of the IR must be a priority. Ideally, we will always be able to preserve that compatibility, forever staying on version 1.x, so that the library ecosystem does not need to be rebuilt.

We hope that we will be able to ensure the continuity of Scala.js for many years to come.

---

[1] https://github.com/tc39/proposal-numeric-separator
[2] https://github.com/tc39/proposal-weakrefs
[3] https://github.com/tc39/proposal-import-meta

Appendix A

Scala.js IR Specification Reference

In this appendix, we give the syntax and typing rules of the Scala.js IR, without any associated explanations. A step-by-step introduction can be found in Chapter 3, whereas this appendix provides the full reference. The specification for the run-time semantics can be found online at http://sebastien.doeraene.be/thesis/sjsir-semantics/.

A.1 Syntax

A.1.1 Syntax of types

IR type-refs

```
 TR  :=  CR | ETR[]
 CR  :=  C | PCR | void | null | nothing
ETR  :=  C | PCR | ETR[]
PCR  :=  boolean | char | byte | short | int | long | float | double
```

IR types

```
 T  :=  void | any | nothing | null | PT | C | ETR[]
PT  :=  boolean | char | byte | short | int | long | float | double
    |   string | undef
```

A.1.2 Syntax of method names

$$
\begin{array}{lll}
\texttt{m} & \texttt{:=} & \texttt{x(}\overline{\texttt{TR}}\texttt{)TR} \qquad \text{(method name)} \\
\texttt{k} & \texttt{:=} & \texttt{<init>(}\overline{\texttt{TR}}\texttt{)} \quad \text{(constructor name)} \\
\texttt{p} & \texttt{:=} & \texttt{x(}\overline{\texttt{TR}}\texttt{)} \qquad \text{(reflective proxy name)}
\end{array}
$$

A.1.3 Syntax of class and member declarations

```
  CD  :=  [ CC ] CK C [ extends C [ via t ] ] [ implements C ] [ NLS ] { CM }
  CC  :=  x: T
  CK  :=  class
      |   module class
      |   interface
      |   abstract js type
      |   js class
      |   js module class
      |   native js class
      |   native js module class
 NLS  :=  loadfrom global:x[SL]
      |   loadfrom import(SL)[SL] [ fallback global:x[SL] ]
  CM  :=  static (val|var) x: T
      |   static def m([var] x: T): T = t
      |   (val|var) x: T
      |   def k([var] x: T) { t }
      |   def m([var] x: T): T = t
      |   def m(x: T): T
      |   [ static ] (val|var) [t]: T
      |   [ static ] def [t]([var] x: any [ [var] ...x: any ]): any = t
      |   [ static ] prop [t] [ get() = t ] [ set([var] x: any) { t } ]
      |   export top class <string literal>
      |   export top module <string literal>
      |   export top static def <string literal>(
                  [var] x: any [ [var] ...x: any ]): any = t
      |   export top static field x as <string literal>
```

A.1.4 Syntax of terms

```
t  :=  val x:T = t; t | var x:T = t; t | t; t
   |   this
   |   x | x = t
   |   mod:C | mod:C = this
   |   skip | debugger
   |   if[T] (t) t else t
   |   while (t) { t }
   |   do { t } while (t)
   |   α[T]: { t }
   |   return@α t
   |   for (val x in t) { t }
   |   try[T] { t } catch (x) { t } | try { t } finally { t }
   |   throw t
   |   match[T] (t) { case <int literal> | ⋯ | <int literal> => t
                      case _ => t }
   |   new C.k(t̄)
   |   t.f | t.f = t
   |   C::f | C::f = t
   |   t.m(t̄) | t.C::m(t̄) | C::m(t̄) | t.p(t̄)
   |   !t | (T)t | t binop t
   |   new ETR[t̄] | new ETR[] { t̄ }
   |   t.arr::length | t.arr::[t] | t.arr::[t] = t
   |   t.isInstanceOf[TR] | t.asInstanceOf[TR]
   |   <getClass>(t)
   |   <linking-info>
   |   new t(t̄)
   |   t[t] | t[t] = t | delete t[t]
   |   t([...]t) | t[t]([...]t)
   |   super(t)::t[t] | super(t)::t[t] = t
   |   super(t)::t[t]([...]t) | super([...]t)
   |   jsunop t | t jsbinop t
   |   constructorOf[C]
   |   [[...]t] | {[t]: t}
   |   global:x
   |   null | <boolean literal> | <char literal> | <byte literal>
   |   <short literal> | <int literal> | <long literal> | <float literal>
   |   <double literal> | <string literal> | undefined
   |   arrow-lambda<x:T=t>([var] [...] x: any) = t
   |   function-lambda<x:T=t>([var] [...] x: any) = t
   |   createjsclass[C](t̄)
```

```
 binop  :=  === | !== | ==[T] | !=[T] | <[T] | <=[T] | >[T] | >=[T]
        |  +[T] | -[T] | *[T] | /[T] | %[T]
        |  |[T] | &[T] | ^[T] | <<[T] | >>[T] | >>>[T]
jsunop  :=  + | - | ~ | !
jsbinop :=  + | - | * | / | % | | | & | ^ | << | >> | >>>
        |  < | <= | > | >= | || | && | in | instanceof
```

A.2 Typing rules

A.2.1 Subclass relationship

$$C < C \quad \text{(C-REFL)} \qquad C < \text{Object} \; \text{(C-OBJ)} \qquad \frac{C < D \quad D < E}{C < E} \; \text{(C-TRANS)}$$

$$\frac{[\; \overline{CC} \;] \; CK \; C \; \text{extends} \; D \; [\; \text{via} \; t \;] \; [\; \text{implements} \; \overline{I} \;] \; [\; NLS \;] \; \{\; \overline{CM} \;\}}{C < D} \; \text{(C-EXTENDS)}$$

$$\frac{I \in \overline{I} \quad [\; \overline{CC} \;] \; CK \; C \; [\; \text{extends} \; D \; [\; \text{via} \; t \;] \;] \; \text{implements} \; \overline{I} \; [\; NLS \;] \; \{\; \overline{CM} \;\}}{C < I} \; \text{(C-IMPLEMENTS)}$$

Although the definition of $<$ is not algorithmic, its domain is finite (it is the set of declared classes in the program) and can therefore be easily precomputed.

A.2.2 Helper functions

Fundamental properties of classes

$$\frac{[\; \overline{CC} \;] \; CK \; C \; [\; \text{extends} \; D \; [\; \text{via} \; t \;] \;] \; [\; \text{implements} \; \overline{I} \;] \; [\; NLS \;] \; \{\; \overline{CM} \;\}}{\text{clskind}(C) = CK} \; \text{(CLS-KIND)}$$

$$\frac{\overline{x{:}T} \; CK \; C \; [\; \text{extends} \; D \; [\; \text{via} \; t \;] \;] \; [\; \text{implements} \; \overline{I} \;] \; [\; NLS \;] \; \{\; \overline{CM} \;\}}{\text{clsenv}(C) = \varnothing, \overline{x}{:}\overline{T}} \; \text{(CLS-ENV-SOME)}$$

$$\frac{CK \; C \; [\; \text{extends} \; D \; [\; \text{via} \; t \;] \;] \; [\; \text{implements} \; \overline{I} \;] \; [\; NLS \;] \; \{\; \overline{CM} \;\}}{\text{clsenv}(C) = \varnothing} \; \text{(CLS-ENV-NONE)}$$

$$\frac{\texttt{clskind(C} \in \{\texttt{class,module class,interface}\}}{\texttt{istype(C)}} \quad \text{(CLS-ISTYPE)}$$

$$\frac{\texttt{clskind(C} \notin \{\texttt{class,module class,interface}\}}{\texttt{isjscls(C)}} \quad \text{(CLS-ISJS)}$$

$$\frac{\texttt{CK C [extends D [via t]] [implements } \overline{\texttt{I}} \texttt{] [NLS] \{ } \overline{\texttt{CM}} \texttt{ \}}}{\texttt{istoplevel(C)}} \quad \text{(CLS-ISTOPLEVEL)}$$

Type of a class

$$\texttt{clstp(O)} = \texttt{any} \text{ (CT-OBJECT)} \qquad \frac{\texttt{istype(C)} \quad \texttt{C} \neq \texttt{O}}{\texttt{clstp(C)} = \texttt{C}} \qquad \frac{\texttt{isjscls(C)}}{\texttt{clstp(C)} = \texttt{any}} \text{ (CT-JS)}$$
$$\text{(CT-TYPED)}$$

Element type of an array

$$\texttt{arreltp(C[])} = \texttt{clstp(C)} \qquad \texttt{arreltp(PCR[])} = \texttt{PCR} \qquad \texttt{arreltp(ETR[][])} = \texttt{ETR[]}$$
$$\text{(AET-C)} \qquad\qquad\qquad \text{(AET-PCR)} \qquad\qquad\qquad \text{(AET-PCR)}$$

Type of a type reference

$$\texttt{trtp(C)} = \texttt{clstp(C)} \quad \text{(TRTP-CLS)} \qquad \frac{\texttt{TR} \neq \texttt{C}}{\texttt{trtp(TR)} = \texttt{TR}} \text{ (TRTP-NONCLS)}$$

Type of method names

$$\texttt{mtype(x(}\overline{\texttt{TR}}\texttt{)TR1)} = (\texttt{trtp(}\overline{\texttt{TR}}\texttt{))}\,\texttt{trtp(TR1)} \qquad \texttt{ktype(<init>(}\overline{\texttt{TR}}\texttt{))} = (\texttt{trtp(}\overline{\texttt{TR}}\texttt{))}$$
$$\text{(METH-TYPE)} \qquad\qquad\qquad\qquad \text{(CTOR-TYPE)}$$

$$\texttt{ptype(x(}\overline{\texttt{TR}}\texttt{))} = (\texttt{trtp(}\overline{\texttt{TR}}\texttt{))}\,\texttt{any}$$
$$\text{(REFLPROXY-TYPE)}$$

Boxed class of primitive types

$$\begin{aligned}
\texttt{boxedcl(boolean)} &= \texttt{jl\_Boolean} & \texttt{boxedcl(char)} &= \texttt{jl\_Character} \\
\texttt{boxedcl(byte)} &= \texttt{jl\_Byte} & \texttt{boxedcl(short)} &= \texttt{jl\_Short} \\
\texttt{boxedcl(int)} &= \texttt{jl\_Integer} & \texttt{boxedcl(long)} &= \texttt{jl\_Long} & \text{(BOXEDCL)} \\
\texttt{boxedcl(float)} &= \texttt{jl\_Float} & \texttt{boxedcl(double)} &= \texttt{jl\_Double} \\
\texttt{boxedcl(string)} &= \texttt{jl\_String} & \texttt{boxedcl(undef)} &= \texttt{sr\_BoxedUnit}
\end{aligned}$$

Appendix A. Scala.js IR Specification Reference

Representative class of a type

Note: tpcls is not defined for `null` nor `nothing`.

$$\text{tpcls(any)} = \texttt{O} \qquad \text{(TC-ANY)} \qquad\qquad \frac{\text{istype(C)}}{\text{tpcls(C)} = \texttt{C}} \quad \text{(TC-CLS)}$$

$$\text{tpcls(PT)} = \text{boxedcl(PR)} \quad \text{(TC-PRIM)} \qquad\qquad \text{tpcls(ETR[])} = \texttt{O} \qquad \text{(TC-ARR)}$$

Set of static fields of a class

$$\frac{\texttt{static val x: T} \in \overline{\text{CM}}}{\texttt{x:T} \in \text{stfields}(\overline{\text{CM}})} \qquad \text{(CF-CMsStValField)}$$

$$\frac{\texttt{static var x: T} \in \overline{\text{CM}}}{\texttt{var x:T} \in \text{stfields}(\overline{\text{CM}})} \qquad \text{(CF-CMsStVarField)}$$

$$\frac{\texttt{[}\overline{\text{CC}}\texttt{] CK [C extends D [via t]] [implements }\overline{\text{I}}\texttt{] [NLS] \{ }\overline{\text{CM}}\texttt{ \}}}{\text{stfields(C)} = \text{stfields(D)}}$$
$$\text{(CF-ClassStFields)}$$

Set of instance fields of a Scala class

$$\frac{\texttt{val x: T} \in \overline{\text{CM}}}{\texttt{x:T} \in \text{fields}(\overline{\text{CM}})} \qquad \text{(CF-CMsValField)}$$

$$\frac{\texttt{var x: T} \in \overline{\text{CM}}}{\texttt{var x:T} \in \text{fields}(\overline{\text{CM}})} \qquad \text{(CF-CMsVarField)}$$

$$\frac{\begin{array}{c}\texttt{[}\overline{\text{CC}}\texttt{] CK C [implements }\overline{\text{I}}\texttt{] [NLS] \{ }\overline{\text{CM}}\texttt{ \}} \\ \text{CK} \in \{\texttt{class,module class}\}\end{array}}{\text{fields(C)} = \text{fields}(\overline{\text{CM}})} \ \text{(CF-TopClassFields)}$$

$$\frac{\begin{array}{c}\texttt{[}\overline{\text{CC}}\texttt{] CK C extends D [via t] [implements }\overline{\text{I}}\texttt{] [NLS] \{ }\overline{\text{CM}}\texttt{ \}} \\ \text{CK} \in \{\texttt{class,module class}\}\end{array}}{\text{fields(C)} = \text{fields(D)} \cup \text{fields}(\overline{\text{CM}})}$$
$$\text{(CF-ClassFields)}$$

Set of constructors of a Scala class

$$\frac{\texttt{def k(}\overline{\texttt{[var] x: T}}\texttt{) \{ t \}} \in \overline{\text{CM}}}{\texttt{k} \in \text{ctors}(\overline{\text{CM}})} \qquad \text{(CF-CMsCtors)}$$

144

$$[\ \overline{\text{CC}} \ ] \ \text{CK} \ \text{C} \ [\ \text{extends} \ \text{D} \ [\ \text{via} \ \text{t} \ ] \ ] \ [\ \text{implements} \ \overline{\text{I}} \ ] \ [\ \text{NLS} \ ] \ \{ \ \overline{\text{CM}} \ \}$$
$$\text{CK} \in \{\text{class}, \text{module class}\}$$

$$\text{ctors(C)} = \text{ctors}(\overline{\text{CM}})$$

(CF-CLASSCTORS)

Set of static methods of a class

$$\frac{\texttt{static def m(}\overline{\texttt{[var] x: T}}\texttt{): T1 = t} \in \overline{\text{CM}}}{\text{m} \in \text{stmethods}(\overline{\text{CM}})}$$

(CF-CMSSTMETHS)

$$[\ \overline{\text{CC}} \ ] \ \text{CK} \ \text{C} \ [\ \text{extends} \ \text{D} \ [\ \text{via} \ \text{t} \ ] \ ] \ [\ \text{implements} \ \overline{\text{I}} \ ] \ [\ \text{NLS} \ ] \ \{ \ \overline{\text{CM}} \ \}$$

$$\text{stmethods(C)} = \text{stmethods}(\overline{\text{CM}})$$

(CF-CLASSSTMETHS)

Set of concrete instance methods of a Scala class or interface

$$\frac{\texttt{def m(}\overline{\texttt{[var] x: T}}\texttt{): T1 = t} \in \overline{\text{CM}}}{\text{m} \in \text{ccmeths}(\overline{\text{CM}})}$$

(CF-CMSCCMETHS)

$$[\ \overline{\text{CC}} \ ] \ \text{CK} \ \text{C} \ \text{extends} \ \text{D} \ [\ \text{via} \ \text{t} \ ] \ [\ \text{implements} \ \overline{\text{I}} \ ] \ [\ \text{NLS} \ ] \ \{ \ \overline{\text{CM}} \ \}$$
$$\text{CK} \in \{\text{class}, \text{module class}, \text{interface}\}$$

$$\text{ccmeths(C)} \supseteq \text{ccmeths(D)}$$

(CF-CLASSCCMETHSSUP)

$$\text{I} \in \overline{\text{I}} \qquad [\ \overline{\text{CC}} \ ] \ \text{CK} \ \text{C} \ [\ \text{extends} \ \text{D} \ [\ \text{via} \ \text{t} \ ] \ ] \ \text{implements} \ \overline{\text{I}} \ [\ \text{NLS} \ ] \ \{ \ \overline{\text{CM}} \ \}$$
$$\text{CK} \in \{\text{class}, \text{module class}, \text{interface}\}$$

$$\text{ccmeths(C)} \supseteq \text{ccmeths(I)}$$

(CF-CLASSCCMETHSINTF)

$$[\ \overline{\text{CC}} \ ] \ \text{CK} \ \text{C} \ [\ \text{extends} \ \text{D} \ [\ \text{via} \ \text{t} \ ] \ ] \ [\ \text{implements} \ \overline{\text{I}} \ ] \ [\ \text{NLS} \ ] \ \{ \ \overline{\text{CM}} \ \}$$
$$\text{CK} \in \{\text{class}, \text{module class}, \text{interface}\}$$

$$\text{ccmeths(C)} \supseteq \text{ccmeths}(\overline{\text{CM}})$$

(CF-CLASSCCMETHSOWN)

Set of instance methods of a Scala class or interface

Including abstract methods and concrete methods.

$$\frac{\texttt{def m(}\overline{\texttt{x: T}}\texttt{): T1} \in \overline{\text{CM}}}{\text{m} \in \text{methods}(\overline{\text{CM}})}$$

(CF-CMSABSMETHS)

$$\frac{\text{m} \in \text{ccmeths}(\overline{\text{CM}})}{\text{m} \in \text{methods}(\overline{\text{CM}})}$$

(CF-CMSMETHS)

145

$$\frac{\begin{array}{c} \texttt{[} \overline{\texttt{CC}} \texttt{] CK C extends D [via t] [implements } \overline{\texttt{I}} \texttt{] [NLS] \{ } \overline{\texttt{CM}} \texttt{ \}} \\ \texttt{CK} \in \{\texttt{class},\texttt{module class},\texttt{interface}\} \end{array}}{\text{methods(C)} \supseteq \text{methods(D)}}$$
$$\text{(CF-CLASSMETHSSUP)}$$

$$\frac{\texttt{I} \in \overline{\texttt{I}} \quad \begin{array}{c} \texttt{[} \overline{\texttt{CC}} \texttt{] CK C [extends D [via t]] implements } \overline{\texttt{I}} \texttt{ [NLS] \{ } \overline{\texttt{CM}} \texttt{ \}} \\ \texttt{CK} \in \{\texttt{class},\texttt{module class},\texttt{interface}\} \end{array}}{\text{methods(C)} \supseteq \text{methods(I)}}$$
$$\text{(CF-CLASSMETHSINTF)}$$

$$\frac{\begin{array}{c} \texttt{[} \overline{\texttt{CC}} \texttt{] CK C [extends D [via t]] [implements } \overline{\texttt{I}} \texttt{] [NLS] \{ } \overline{\texttt{CM}} \texttt{ \}} \\ \texttt{CK} \in \{\texttt{class},\texttt{module class},\texttt{interface}\} \end{array}}{\text{methods(C)} \supseteq \text{methods(}\overline{\texttt{CM}}\text{)}}$$
$$\text{(CF-CLASSMETHSOWN)}$$

A.2.3 Subtyping relationship

Subclassing of array element type-refs

$$\texttt{PCR} <_{[]} \texttt{PCR} \quad \text{(ES-PRIM)}$$

$$\frac{\texttt{C} < \texttt{D}}{\texttt{C} <_{[]} \texttt{D}} \quad \text{(ES-SUBCL)}$$

$$\frac{\texttt{ETR1} <_{[]} \texttt{ETR2}}{\texttt{ETR1 []} <_{[]} \texttt{ETR2 []}} \quad \text{(ES-SUBETR)}$$

$$\frac{\texttt{java.lang.Cloneable} < \texttt{D}}{\texttt{ETR1 []} <_{[]} \texttt{D}}$$
$$\text{(ES-ARRCLONEABLE)}$$

$$\frac{\texttt{java.io.Serializable} < \texttt{D}}{\texttt{ETR1 []} <_{[]} \texttt{D}}$$
$$\text{(ES-ARRSERIALIZABLE)}$$

The $<_{[]}$ relationship is not technically algorithmic because of the overlap between the rules ES-ARRCLONEABLE and ES-ARRSERIALIZABLE. However, a practical implementation can still be easily derived, by trying first one rule then the other. From this almost-algorithmic definition, we can derive reflexivity and transitivity.

Lemma 2 (Reflexivity of $<_{[]}$). *For any array element type-ref ETR, ETR* $<_{[]}$ *ETR.*

Proof. By structural induction and case analysis on ETR:

- Case PCR: immediate from ES-PRIM

- Case C: follows from ES-SUBCL using C-REFL, i.e., C $<$ C.

- Case ETR1 []: follows from ES-SUBETR using the induction hypothesis, i.e., ETR1 $<_{[]}$ ETR1.

$\square$

Lemma 3 (Transitivity of $<_{[]}$)**.** *For any three array element type-refs ETR1, ETR2 and ETR3, if ETR1 $<_{[]}$ ETR2 and ETR2 $<_{[]}$ ETR3, then ETR1 $<_{[]}$ ETR3.*

Proof. By structural induction and case analysis on ETR1:

- Case PCR1: PCR1 $<_{[]}$ ETR2 must come from ES-PRIM, which means that ETR2 = PCR1. Similarly, we have ETR3 = ETR2 = PCR1, from which we derive PCR1 $<_{[]}$ ETR3 using ES-PRIM.

- Case C1: C1 $<_{[]}$ ETR2 must come from ES-SUBCL, which means that ETR2 = C2 such that C1 $<$ C2. Similarly, we must have ETR3 = C3 such that C2 $<$ C3. Using C-TRANS, we have C1 $<$ C3, from which we derive C1 $<_{[]}$ C3 using ES-SUBCL.

- Case ETR11[]: by case analysis on the rule used to derive ETR11[] $<_{[]}$ ETR2:

    - Case ES-ARRCLONEABLE: ETR2 = C2. In this case, C2 $<_{[]}$ ETR3 must come from ES-SUBCL with ETR3 = C3 and C2 $<$ C3. By C-TRANS, we have Cloneable $<$ C3, and therefore ETR11[] $<_{[]}$ C3 by ES-ARRCLONEABLE.

    - Case ES-ARRSERIALIZABLE: similar.

    - Case ES-SUBETR: ETR2 = ETR21[], and we decompose again by case analysis on the rule used to derive ETR21 $<_{[]}$ ETR3:

        * Case ES-ARRCLONEABLE: ETR3 = C3 with `java.lang.Cloneable` $<$ C3, from which we derive ETR11[] $<_{[]}$ C3 with ES-ARRCLONEABLE.

        * Case ES-ARRSERIALIZABLE: similar.

        * Case ES-SUBETR: ETR3 = ETR31[], and we conclude by the induction hypothesis.

$\square$

Subtyping

$$T <: \text{void} \qquad \text{(S-VOID)}$$

$$\frac{\text{boxedcl(PT)} < C}{PT <: C} \quad \text{(S-PRIMCL)}$$

$$\frac{T \neq \text{void}}{T <: \text{any}} \quad \text{(S-ANY)}$$

$$\text{null} <: \text{null} \quad \text{(S-NULLREFL)}$$

$$\frac{T \notin \{\text{void}, \text{any}\}}{\text{nothing} <: T} \quad \text{(S-NOTHING)}$$

$$\frac{\text{istype(C)}}{\text{null} <: C} \quad \text{(S-NULLCL)}$$

$$\frac{C < D \qquad \text{istype(C)} \qquad \text{istype(D)}}{C <: D} \quad \text{(S-SUBCL)}$$

$$\text{null} <: ETR[] \quad \text{(S-NULLARR)}$$

$$PT <: PT \quad \text{(S-PRIMREFL)}$$

$$\frac{ETR1[] <_{[]} D}{ETR1[] <: D} \quad \text{(S-ARRCL)}$$

147

$$\frac{\texttt{ETR1[]} <_{[]} \texttt{ETR2[]}}{\texttt{ETR1[]} <: \texttt{ETR2[]}} \text{ (S-ARRARR)}$$

The above definition is algorithmic. We can derive reflexivity and transitivity as follows. Note that the definition of reflexivity is slightly tricky, since we must only cover *valid types*. Recall that class names `C` where `C` is a JavaScript types are not valid types, even if they belong to the syntactic category of types.

Theorem 4 (Reflexivity of `<:`). *For any valid type T, i.e., where T = C implies* istype(*C*)*, T <: T.*

Proof. Straightforward by case analysis on T:

- Cases `void`, `any`, `nothing`, `null` and `PT`: immediate by S-VOID, S-ANY, S-NOTHING, S-NULLREFL and S-PRIMREFL, respectively.

- Case `C`: since `T` is a valid type, we have istype(`C`), and therefore we have `C <: C` by S-SUBCL and C-REFL ($C \prec C$).

- Case `ETR[]`: immediate from S-ARRARR and reflexivity of $<_{[]}$.

$\square$

Theorem 5 (Transitivity of `<:`). *For any three types T1, T2 and T3, if T1 <: T2 and T2 <: T3, then T1 <: T3.*

Proof. We start by case analysis on T3 to take care of the cases T3 = `void` and T3 = `any`.

If T3 = `void`, we trivially conclude with S-VOID.

If T3 = `any`, T2 `<:` T3 must have been derived from S-ANY which means that T2 $\neq$ `void`. In turn, this means that T1 `<:` T2 cannot have been derived from S-VOID, and none of the other rules allow T1 = `void`, from which we derive that T1 $\neq$ `void`. We conclude with S-ANY.

Otherwise, T3 $\notin \{\texttt{void}, \texttt{any}\}$ (which we refer to as †) and we proceed by case analysis on the last rule used to derive T1 `<:` T2:

- Case S-VOID: T2 = `void`, hence T2 `<:` T3 must also be derived from S-VOID, which contradicts †.

- Case S-ANY: T2 = `any`, hence T2 `<:` T3 must have been derived either from S-VOID or S-ANY, both of which contradict †.

- Case S-NOTHING: T1 = `nothing`, so we conclude with S-NOTHING using †.

- Case S-SUBCL: T1 = C1 and T2 = C2 with $C1 \prec C2$ and istype(C1), and since †, the last rule for T2 `<:` T3 must also be S-SUBCL, from which T3 = C3 with $C2 \prec C3$ and istype(C3). We use C-TRANS to get $C1 \prec C3$, from which we conclude with S-SUBCL.

- Case S-PRIMCL: similar (replacing C1 by boxedcl(PT), and using S-PRIMCL to conclude).

- Case S-PRIMREFL: T1 = T2, hence T1 <: T3 is trivially obtained from T2 <: T3.

- Case S-NULLREFL: similar.

- Case S-NULLCL: by †, the last rule used to derive T2 <: T3 must have been S-SUBCL, from which T3 = C3 and istype(C3). We conclude with S-NULLCL.

- Case S-NULLARR: similar, although we have two possible last rules for T2 <: T3, namely S-ARRCL or S-ARRARR, so we conclude with S-NULLCL or S-NULLARR, respectively.

- Case S-ARRCL: T1 = ETR1[] and T2 = C2 with ETR1[] $<_{[]}$ C2. By †, the last rule used for T2 <: T3 was S-SUBCL, from which T3 = C3 with C2 < C3. We use ES-SUBCL to obtain C2 $<_{[]}$ C3, then transitivy of $<_{[]}$ to obtain ETR1[] $<_{[]}$ C3. We conclude with S-ARRCL.

- Case S-ARRARR: T1 = ETR1[] and T2 = ETR2[] with ETR1[] $<_{[]}$ ETR2[]. By †, the last rule used for T2 <: T3 was either S-ARRCL with T3 = C3 and ETR2[] $<_{[]}$ C3, from which we conclude with S-ARRCL and transitivity of $<_{[]}$, or S-ARRARR with T3 = ETR3[] and ETR2[] $<_{[]}$ ETR3[], from which we conclude with S-ARRARR and transitivity of $<_{[]}$.

$\square$

A.2.4 Typing of members and class definitions

Typing of class members

$$\frac{T \neq void}{\texttt{static (val|var) x: T OK IN C}} \quad \text{(ST FIELD-OK)}$$

$$\frac{mtype(m) = (\overline{T})T1 \qquad \varnothing, \overline{[var]x:T} \vdash t:T1}{\texttt{static def m(}\overline{[var]\ x:\ T}\texttt{): T1 = t OK IN C}} \quad \text{(ST METH-OK)}$$

$$\frac{clskind(C) \in \{\texttt{class,module class,js class,js module class}\} \quad \nexists\ T1 \in PT\ s.t.\ boxedcl(T1) < C \qquad T \neq void}{\texttt{(val|var) x: T OK IN C}} \quad \text{(FIELD-OK)}$$

$$\frac{clskind(C) \in \{\texttt{class,module class}\} \quad ktype(k) = (\overline{T}) \qquad \varnothing, \mathscr{C}(C), this:C, \overline{[var]x:T} \vdash t:T1}{\texttt{def k(}\overline{[var]\ x:\ T}\texttt{) \{ t \} OK IN C}} \quad \text{(CTOR-OK)}$$

$$\frac{clskind(C) \in \{\texttt{class,module class,interface}\} \quad mtype(m) = (\overline{T})T1 \qquad \varnothing, this:C, \overline{[var]x:T} \vdash t:S1 \qquad S1 <: T1}{\texttt{def m(}\overline{[var]\ x:\ T}\texttt{): T1 = t OK IN C}} \quad \text{(METH-OK)}$$

149

$$\frac{\begin{array}{c} \text{clskind(C)} \in \{\texttt{class},\texttt{module class},\texttt{interface}\} \\ \text{mtype(m)} = (\overline{\texttt{T}})\texttt{T1} \end{array}}{\texttt{def m}(\overline{\texttt{x: T}})\texttt{: T1 OK IN C}} \quad \text{(ABSMETH-OK)}$$

$$\frac{\begin{array}{c} \text{clskind(C)} \in \{\texttt{class},\texttt{module class}\} \\ \varnothing,\texttt{this:C},\overline{\texttt{[var]x:any}} \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \end{array}}{\texttt{def [<string literal>]}(\texttt{[var] [}\dots\texttt{]x: any})\texttt{: any = t1 OK IN C}} \quad \text{(EXPMETH-OK)}$$

$$\frac{\begin{array}{c} \text{clskind(C)} \in \{\texttt{class},\texttt{module class}\} \\ [\varnothing,\texttt{this:C} \vdash \texttt{t1:S1} \quad \texttt{S1 <: any}] \\ [\varnothing,\texttt{this:C},\texttt{[var]x:any} \vdash \texttt{t2:T2}] \end{array}}{\texttt{prop [<string literal>] [get() = t1] [set([var] x: any) \{ t2 \}] OK IN C}}$$
$$\text{(EXPPROP-OK)}$$

$$\frac{\begin{array}{c} \text{clskind(C)} \in \{\texttt{js class},\texttt{js module class}\} \\ \texttt{T} \neq \texttt{void} \quad \text{clsenv(C)} \vdash \texttt{t:S} \quad \texttt{S <: any} \end{array}}{\texttt{[static] (val|var) [t]: T OK IN C}} \quad \text{(JSFIELD-OK)}$$

$$\frac{\begin{array}{c} \text{clskind(C)} \in \{\texttt{js class},\texttt{js module class}\} \\ \text{clsenv(C)},\overline{\texttt{[var]x:any}} \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \\ \text{clsenv(C)},\overline{\texttt{[var]x:any}} \vdash \overline{\texttt{t}}:\overline{\texttt{S}} \quad \overline{\texttt{S}} \texttt{ <: any} \\ \text{clsenv(C)},\texttt{this:any},\overline{\texttt{[var]x:any}} \vdash \texttt{t2:S2} \quad \texttt{S2 <: any} \end{array}}{\begin{array}{c} \texttt{def ["constructor"]}(\texttt{[var] [}\dots\texttt{]x: any})\texttt{: any = \{} \\ \texttt{t1; super(}\overline{\texttt{[}\dots\texttt{]t}}\texttt{); t2 \} OK IN C} \end{array}} \quad \text{(JSCTOR-OK)}$$

$$\frac{\begin{array}{c} \text{clskind(C)} \in \{\texttt{js class},\texttt{js module class}\} \\ \texttt{t1} \neq \texttt{"constructor"} \quad \text{clsenv(C)} \vdash \texttt{t1:S} \quad \texttt{S <: any} \\ \text{clsenv(C)},\texttt{this:any},\overline{\texttt{[var]x:any}} \vdash \texttt{t2:S2} \quad \texttt{S2 <: any} \end{array}}{\texttt{[static] def [t1](}\texttt{[var] [}\dots\texttt{]x: any})\texttt{: any = t2 OK IN C}} \quad \text{(JSMETH-OK)}$$

$$\frac{\begin{array}{c} \text{clskind(C)} \in \{\texttt{js class},\texttt{js module class}\} \\ \text{clsenv(C)} \vdash \texttt{t1:S} \quad \texttt{S <: any} \\ [\text{clsenv(C)},\texttt{this:any} \vdash \texttt{t2:S2} \quad \texttt{S2 <: any}] \\ [\text{clsenv(C)},\texttt{this:any},\texttt{[var]x:any} \vdash \texttt{t3:T3}] \end{array}}{\texttt{[static] prop [t1] [get() = t2] [set([var] x: any) \{ t3 \}] OK IN C}}$$
$$\text{(JSPROP-OK)}$$

$$\frac{\text{clskind(C)} = \texttt{js class}}{\texttt{export top class <string literal> OK IN C}} \quad \text{(TL-CLS-OK)}$$

$$\frac{\text{clskind(C)} \in \{\texttt{module class},\texttt{js module class}\}}{\texttt{export top module <string literal> OK IN C}} \quad \text{(TL-MOD-OK)}$$

$$\frac{\begin{array}{c} \text{clskind(C)} \in \{\texttt{class,module class,js class,js module class}\} \\ \varnothing, \texttt{this:any}, \overline{\texttt{[var]x:any}} \vdash \texttt{t2:S2} \qquad \texttt{S2 <: any} \end{array}}{\texttt{export top static def <string lit>}(\overline{\texttt{[var] [...]x: any}})\texttt{: any = t2 OK IN C}} \text{(TL-FUN-OK)}$$

$$\frac{\begin{array}{c} \text{clskind(C)} \in \{\texttt{class,module class,js class,js module class}\} \\ \texttt{x:any} \in \text{stfields(C)} \end{array}}{\texttt{export top static field x as <string literal> OK IN C}} \text{(TL-FIELD-OK)}$$

Typing of classes

$$\frac{\overline{\texttt{CM}} \texttt{ OK IN C} \qquad \texttt{CK} \in \{\texttt{class,module class}\}}{\texttt{[} \overline{\texttt{CC}} \texttt{] CK C [implements } \overline{\texttt{I}} \texttt{] [NLS] \{ } \overline{\texttt{CM}} \texttt{ \} OK}} \text{(CLS-OKTOP)}$$

$$\frac{\overline{\texttt{CM}} \texttt{ OK IN C} \qquad \texttt{CK} \in \{\texttt{class,module class}\} \qquad \text{fields}(\overline{\texttt{CM}}) \cap \text{fields(D)} = \varnothing}{\texttt{[} \overline{\texttt{CC}} \texttt{] CK C extends D [via t] [implements } \overline{\texttt{I}} \texttt{] [NLS] \{ } \overline{\texttt{CM}} \texttt{ \} OK}} \text{(CLS-OKWITHFLDS)}$$

$$\frac{\overline{\texttt{CM}} \texttt{ OK IN C} \qquad \texttt{CK} \notin \{\texttt{class,module class}\}}{\texttt{[} \overline{\texttt{CC}} \texttt{] CK C [extends D [via t]] [implements } \overline{\texttt{I}} \texttt{] [NLS] \{ } \overline{\texttt{CM}} \texttt{ \} OK}} \text{(CLS-OKNOFLDS)}$$

A.2.5 Typing of terms

Variables and fields

$$\frac{\texttt{T1} \neq \texttt{void} \qquad \Gamma \vdash \texttt{t1:S1} \qquad \Gamma, \texttt{x:T1} \vdash \texttt{t2:T2} \qquad \texttt{S1 <: T1}}{\Gamma \vdash \texttt{val x:T1 = t1; t2 : T2}} \text{(T-VALDEF)}$$

$$\frac{\texttt{T1} \neq \texttt{void} \qquad \Gamma \vdash \texttt{t1:S1} \qquad \Gamma, \texttt{var x:T1} \vdash \texttt{t2:T2} \qquad \texttt{S1 <: T1}}{\Gamma \vdash \texttt{var x:T1 = t1; t2 : T2}} \text{(T-VARDEF)}$$

$$\frac{\Gamma \vdash \texttt{t1:T1} \qquad \Gamma \vdash \texttt{t2:T2} \qquad \texttt{t1} \notin \{\texttt{val...,var...}\}}{\Gamma \vdash \texttt{t1; t2 : T2}} \text{(T-SEQ)}$$

$$\frac{\texttt{this:T} \in \Gamma}{\Gamma \vdash \texttt{this: T}} \text{(T-THISREF)}$$

$$\frac{\texttt{[var]x:T} \in \Gamma}{\Gamma \vdash \texttt{x: T}} \text{(T-VARREF)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{var x:T1} \in \Gamma \quad \texttt{S1 <: T1}}{\Gamma \vdash \texttt{x = t1 : void}} \quad \text{(T-AssignVar)}$$

Control structures

$$\Gamma \vdash \texttt{skip : void} \qquad \text{(T-Skip)}$$

$$\frac{\begin{array}{ccc}\Gamma \vdash \texttt{t1:S1} & \Gamma \vdash \texttt{t2:S2} & \Gamma \vdash \texttt{t3:S3}\\ \texttt{S1 <: boolean} & \texttt{S2 <: T} & \texttt{S3 <: T}\end{array}}{\Gamma \vdash \texttt{if[T] (t1) t2 else t3 : T}} \quad \text{(T-If)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{t1} \neq \texttt{true} \quad \texttt{S1 <: boolean} \quad \texttt{t2:T2}}{\Gamma \vdash \texttt{while (t1) \{ t2 \} : void}} \quad \text{(T-While)}$$

$$\frac{\Gamma \vdash \texttt{t2:T2}}{\Gamma \vdash \texttt{while (true) \{ t2 \} : nothing}} \quad \text{(T-WhileTrue)}$$

$$\frac{\Gamma \vdash \texttt{t1:T1} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: boolean}}{\Gamma \vdash \texttt{do \{ t1 \} while (t2) : void}} \quad \text{(T-DoWhile)}$$

$$\frac{\Gamma, \alpha\texttt{:T} \vdash \texttt{t1:S} \quad \texttt{S <: T}}{\Gamma \vdash (\alpha\texttt{[T]: \{ t1 \}): T}} \quad \text{(T-Labeled)}$$

$$\frac{\Gamma \vdash \texttt{t1:S} \quad \alpha\texttt{:T} \in \Gamma \quad \texttt{S} \neq \texttt{void} \quad \texttt{S <: T}}{\Gamma \vdash \texttt{return@}\alpha \texttt{ t1 : nothing}} \quad \text{(T-Return)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma, \texttt{x:any} \vdash \texttt{t2:T2}}{\Gamma \vdash \texttt{for (val x in t1) \{ t2 \} : void}} \quad \text{(T-ForIn)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \Gamma, \texttt{x:any} \vdash \texttt{t2:S2} \quad \texttt{S1 <: T} \quad \texttt{S2 <: T}}{\Gamma \vdash \texttt{try[T] \{ t1 \} catch (x) \{ t2 \} : T}} \quad \text{(T-TryCatch)}$$

$$\frac{\Gamma \vdash \texttt{t1:T} \quad \Gamma \vdash \texttt{t2:T2}}{\Gamma \vdash \texttt{try \{ t1 \} finally \{ t2 \} : T}} \quad \text{(T-TryFinally)}$$

$$\frac{\Gamma \vdash \texttt{t1:S} \quad \texttt{S <: any}}{\Gamma \vdash \texttt{throw t1 : nothing}} \quad \text{(T-Throw)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: int} \quad \Gamma \vdash \overline{\texttt{t}}\texttt{:}\overline{\texttt{S}} \quad \overline{\texttt{S}} \texttt{ <: T} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: T2}}{\Gamma \vdash \texttt{match[T] (t1) \{ }\overline{\texttt{case IntLit | } \cdots \texttt{ | IntLit => t}}\texttt{ case \_ => t2 \} : T}}$$
$$\text{(T-Match)}$$

$$\Gamma \vdash \texttt{debugger : void} \qquad \text{(T-DEBUGGER)}$$

Operations on Scala classes

$$\frac{\begin{array}{c} \text{clskind(C)} = \texttt{class} \qquad \text{k} \in \text{ctors(C)} \qquad \text{ktype(k)} = (\overline{\text{T}}) \\ \Gamma \vdash \overline{\text{t}} : \overline{\text{S}} \qquad \overline{\text{S}} \texttt{ <: } \overline{\text{T}} \qquad \text{ccmeths(C)} = \text{methods(C)} \end{array}}{\Gamma \vdash \texttt{new C.k(}\overline{\text{t}}\texttt{) : C}} \qquad \text{(T-NEW)}$$

$$\frac{\text{clskind(C)} = \texttt{module class}}{\Gamma \vdash \texttt{mod:C : C}} \qquad \text{(T-LOADMODULE)}$$

$$\frac{\mathscr{C}(\text{C}) \in \Gamma \qquad \texttt{this:T} \in \Gamma}{\Gamma \vdash \texttt{mod:C = this : void}} \qquad \text{(T-STOREMOD)}$$

$$\frac{\Gamma \vdash \texttt{t1:C1} \qquad \texttt{[var]f:T2} \in \text{fields}(\textit{C1})}{\Gamma \vdash \texttt{t1.f : T2}} \qquad \text{(T-FLDREF)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \qquad \texttt{S1 <: null}}{\Gamma \vdash \texttt{t1.f : nothing}} \qquad \text{(T-NULLFLDREF)}$$

$$\frac{\begin{array}{c} \Gamma \vdash \texttt{t1:C1} \qquad \texttt{var f:T2} \in \text{fields(C1)} \\ \Gamma \vdash \texttt{t2:S2} \qquad \texttt{S2 <: T2} \end{array}}{\Gamma \vdash \texttt{t1.f = t2 : void}} \qquad \text{(T-ASSFLD)}$$

$$\frac{\begin{array}{c} \mathscr{C}(\text{C1}) \in \Gamma \qquad \texttt{this:C1} \in \Gamma \qquad \texttt{f:T2} \in \text{fields(C1)} \\ \Gamma \vdash \texttt{t2:S2} \qquad \texttt{S2 <: T2} \end{array}}{\Gamma \vdash \texttt{this.f = t2 : void}} \qquad \text{(T-ASSVALFLD)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \qquad \texttt{S1 <: null} \qquad \Gamma \vdash \texttt{t2:S2} \qquad \texttt{S2 <: any}}{\Gamma \vdash \texttt{t1.f = t2 : void}} \qquad \text{(T-ASSNULLFLD)}$$

$$\frac{\texttt{[var]f:T} \in \text{stfields(C)}}{\Gamma \vdash \texttt{C::f : T2}} \qquad \text{(T-STFLDREF)}$$

$$\frac{\texttt{var f:T1} \in \text{stfields(C)} \qquad \Gamma \vdash \texttt{t1:S1} \qquad \texttt{S1 <: T1}}{\Gamma \vdash \texttt{C::f = t2 : void}} \qquad \text{(T-ASSSTFLD)}$$

$$\frac{\begin{array}{c} \text{m} \in \text{methods(tpcls(T1))} \qquad \text{mtype(m)} = (\overline{\text{T}})\text{R} \\ \Gamma \vdash \texttt{t1:T1} \qquad \texttt{T1} \notin \{\texttt{void}, \texttt{null}, \texttt{nothing}\} \qquad \Gamma \vdash \overline{\text{t}} : \overline{\text{S}} \qquad \overline{\text{S}} \texttt{ <: } \overline{\text{T}} \end{array}}{\Gamma \vdash \texttt{t1.m(}\overline{\text{t}}\texttt{) : R}} \qquad \text{(T-APPLY)}$$

$$\frac{\begin{array}{c} \text{mtype(m)} = (\overline{T})\,R \\ \Gamma \vdash t1 : T1 \qquad T1 \in \{\text{null,nothing}\} \qquad \Gamma \vdash \overline{t} : \overline{S} \qquad \overline{S} <: \overline{T} \end{array}}{\Gamma \vdash t1.m(\overline{t})\ :\ R} \quad \text{(T-ApplyNull)}$$

$$\frac{\begin{array}{c} S1 <: C \qquad m \in \text{ccmeths(C)} \qquad \text{mtype(m)} = (\overline{T})\,R \\ \Gamma \vdash t1 : S1 \qquad \Gamma \vdash \overline{t} : \overline{S} \qquad \overline{S} <: \overline{T} \end{array}}{\Gamma \vdash t1.C::m(\overline{t})\ :\ R} \quad \text{(T-ApplyStly)}$$

$$\frac{\begin{array}{c} m \in \text{stmethods(C)} \qquad \text{mtype(m)} = (\overline{T})\,R \\ \Gamma \vdash \overline{t} : \overline{S} \qquad \overline{S} <: \overline{T} \end{array}}{\Gamma \vdash C::m(\overline{t})\ :\ R} \quad \text{(T-ApplyStatic)}$$

$$\frac{\begin{array}{c} \text{ptype(p)} = (\overline{T})\,\text{any} \\ \Gamma \vdash t1 : S1 \qquad S1 <: \text{any} \qquad \Gamma \vdash \overline{t} : \overline{S} \qquad \overline{S} <: \overline{T} \end{array}}{\Gamma \vdash t1.p(\overline{t})\ :\ \text{any}} \quad \text{(T-ApplyRefl)}$$

Typed unary and binary operators

$$\frac{\Gamma \vdash t : S \qquad S <: \text{boolean}}{\Gamma \vdash \ !t\ :\ \text{boolean}} \quad \text{(T-BoolUnary)}$$

$$\frac{\Gamma \vdash t : S \qquad S <: \text{int}}{\Gamma \vdash (\text{char})t\ :\ \text{char}} \quad \text{(T-ToChar)} \qquad \frac{\Gamma \vdash t : T \qquad T \in \{\text{nothing,int,double}\}}{\Gamma \vdash (\text{long})t\ :\ \text{long}} \quad \text{(T-ToLong)}$$

$$\frac{\Gamma \vdash t : S \qquad S <: \text{int}}{\Gamma \vdash (\text{byte})t\ :\ \text{byte}} \quad \text{(T-ToByte)} \qquad \frac{\Gamma \vdash t : S \qquad S <: \text{double}}{\Gamma \vdash (\text{float})t\ :\ \text{float}} \quad \text{(T-ToFloat)}$$

$$\frac{\Gamma \vdash t : S \qquad S <: \text{int}}{\Gamma \vdash (\text{short})t\ :\ \text{short}} \quad \text{(T-ToShort)} \qquad \frac{\Gamma \vdash t : T \qquad T \in \{\text{nothing,int,float,long}\}}{\Gamma \vdash (\text{double})t\ :\ \text{double}} \quad \text{(T-ToDouble)}$$

$$\frac{\Gamma \vdash t : T \qquad T \in \{\text{nothing,char,byte,short,long,double}\}}{\Gamma \vdash (\text{int})t\ :\ \text{int}} \quad \text{(T-ToInt)}$$

$$\frac{\Gamma \vdash t1 : S1 \qquad S1 <: \text{any} \qquad \Gamma \vdash t2 : S2 \qquad S2 <: \text{any}}{\Gamma \vdash t1\ (===|\,!==)\ t2\ :\ \text{boolean}} \quad \text{(T-EqNe)}$$

$$\frac{\Gamma \vdash t1 : S1 \qquad S1 <: \text{any} \qquad \Gamma \vdash t2 : S2 \qquad S2 <: \text{any}}{\Gamma \vdash t1\ +[\text{string}]\ t2\ :\ \text{string}} \quad \text{(T-StrCat)}$$

$$\frac{\Gamma \vdash t1 : S1 \qquad S1 <: \text{boolean} \qquad \Gamma \vdash t2 : S2 \qquad S2 <: \text{boolean}}{\Gamma \vdash t1\ (==|\,!=|\,|\,|\,\&)[\text{boolean}]\ t2\ :\ \text{boolean}} \quad \text{(T-BoolOp)}$$

$$\frac{\text{T} \in \{\text{int}, \text{long}, \text{float}, \text{double}\} \quad \Gamma \vdash \text{t1:S1} \quad \text{S1 <: T} \quad \Gamma \vdash \text{t2:S2} \quad \text{S2 <: T}}{\Gamma \vdash \text{t1 (+|-|*|/|\%)[T] t2 : T}}$$
(T-ARITHOP)

$$\frac{\text{T} \in \{\text{int}, \text{long}\} \quad \Gamma \vdash \text{t1:S1} \quad \text{S1 <: T} \quad \Gamma \vdash \text{t2:S2} \quad \text{S2 <: T}}{\Gamma \vdash \text{t1 (||\&|\^{})[T] t2 : T}}$$
(T-BITWISEOP)

$$\frac{\text{T} \in \{\text{int}, \text{long}\} \quad \Gamma \vdash \text{t1:S1} \quad \text{S1 <: T} \quad \Gamma \vdash \text{t2:S2} \quad \text{S2 <: int}}{\Gamma \vdash \text{t1 (<<|>>|>>>)[T] t2 : T}}$$
(T-SHIFTOP)

$$\frac{\text{T} \in \{\text{int}, \text{long}, \text{double}\} \quad \Gamma \vdash \text{t1:S1} \quad \text{S1 <: T} \quad \Gamma \vdash \text{t2:S2} \quad \text{S2 <: T}}{\Gamma \vdash \text{t1 (==|!=|<|<=|>|>=)[T] t2 : boolean}}$$
(T-CMP)

Operations on arrays

$$\frac{\overline{t} \text{ is not empty} \quad \Gamma \vdash \overline{\text{t}}:\overline{\text{S}} \quad \overline{\text{S}} \text{ <: int}}{\Gamma \vdash \text{new ETR}[\overline{\text{t}}] \text{ : ETR[]}}$$
(T-NEWARR)

$$\frac{\Gamma \vdash \text{t:ETR[]}}{\Gamma \vdash \text{t.arr::length : int}}$$
(T-ARRLENGTH)

$$\frac{\Gamma \vdash \overline{\text{t}}:\overline{\text{S}} \quad \overline{\text{S}} \text{ <: arreltp(ETR[])}}{\Gamma \vdash \text{new ETR[] \{ } \overline{\text{t}} \text{ \} : ETR[]}}$$
(T-ARRVALUE)

$$\frac{\Gamma \vdash \text{t1:S1} \quad \text{S1 <: null}}{\Gamma \vdash \text{t1.arr::length : int}}$$
(T-NULLARRLENGTH)

$$\frac{\Gamma \vdash \text{t1:ETR1[]} \quad \Gamma \vdash \text{t2:S2} \quad \text{S2 <: int} \quad \text{T = arreltp(ETR1[])}}{\Gamma \vdash \text{t1.arr::[t2] : T}}$$
(T-ARRELEM)

$$\frac{\Gamma \vdash \text{t1:S1} \quad \text{S1 <: null} \quad \Gamma \vdash \text{t2:S2} \quad \text{S2 <: int}}{\Gamma \vdash \text{t1.arr::[t2] : nothing}}$$
(T-NULLARRELEM)

$$\frac{\Gamma \vdash \text{t1:ETR1[]} \quad \Gamma \vdash \text{t2:S2} \quad \text{S2 <: int} \quad \Gamma \vdash \text{t3:S3} \quad \text{S3 <: arreltp(ETR1[])}}{\Gamma \vdash \text{t1.arr::[t2] = t3 : void}}$$
(T-ASSARRELEM)

$$\frac{\Gamma \vdash \text{t1:S1} \quad \text{S1 <: null} \quad \Gamma \vdash \text{t2:S2} \quad \text{S2 <: int} \quad \Gamma \vdash \text{t3:S3} \quad \text{S3 <: any}}{\Gamma \vdash \text{t1.arr::[t2] = t3 : void}}$$
(T-ASSNULLARRELEM)

Instance tests and casts

$$\frac{\Gamma \vdash \text{t1:T1} \quad \text{T1 <: any} \quad \text{istype(C)}}{\Gamma \vdash \text{t1.isInstanceOf[C] : boolean}}$$
(T-ISCLS)

155

Appendix A. Scala.js IR Specification Reference

$$\frac{\Gamma \vdash \texttt{t1:T1} \quad \texttt{T1 <: any}}{\Gamma \vdash \texttt{t1.isInstanceOf[ETR[]] : boolean}} \quad \text{(T-IsArr)}$$

$$\frac{\Gamma \vdash \texttt{t1:T1} \quad \texttt{T1 <: any} \quad \texttt{istype(C)}}{\Gamma \vdash \texttt{t1.asInstanceOf[C] : C}} \quad \text{(T-AsCls)}$$

$$\frac{\Gamma \vdash \texttt{t1:T1} \quad \texttt{T1 <: any}}{\Gamma \vdash \texttt{t1.asInstanceOf[PCR] : PCR}} \quad \text{(T-AsPrim)}$$

$$\frac{\Gamma \vdash \texttt{t1:T1} \quad \texttt{T1 <: any}}{\Gamma \vdash \texttt{t1.asInstanceOf[ETR[]] : ETR[]}} \quad \text{(T-AsArr)}$$

$$\frac{\Gamma \vdash \texttt{t1:T1} \quad \texttt{T1 <: any}}{\Gamma \vdash \texttt{<getClass>(t1) : jl\_Class}} \quad \text{(T-GetCls)}$$

JavaScript operations

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \overline{\texttt{t}}:\overline{\texttt{S}} \quad \overline{\texttt{S}} \texttt{ <: any}}{\Gamma \vdash \texttt{new t1(}\overline{\texttt{t}}\texttt{) : any}} \quad \text{(T-JSNew)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: any}}{\Gamma \vdash \texttt{t1[t2] : any}} \quad \text{(T-JSSelect)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: any} \quad \Gamma \vdash \texttt{t3:S3} \quad \texttt{S3 <: any}}{\Gamma \vdash \texttt{t1[t2] = t3 : void}}$$

$$\text{(T-JSAssSelect)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: any}}{\Gamma \vdash \texttt{delete t1[t2] : void}} \quad \text{(T-JSDelete)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \overline{\texttt{t}}:\overline{\texttt{S}} \quad \overline{\texttt{S}} \texttt{ <: any}}{\Gamma \vdash \texttt{t1(}\overline{\texttt{[...]t}}\texttt{) : any}} \quad \text{(T-JSFunApply)}$$

$$\frac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: any} \quad \Gamma \vdash \overline{\texttt{t}}:\overline{\texttt{S}} \quad \overline{\texttt{S}} \texttt{ <: any}}{\Gamma \vdash \texttt{t1[t2](}\overline{\texttt{[...]t}}\texttt{) : any}}$$

$$\text{(T-JSMethApply)}$$

$$\frac{\Gamma \vdash \texttt{t0:S0} \quad \texttt{S0 <: any} \quad \Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: any}}{\Gamma \vdash \texttt{super(t0)::t1[t2] : any}}$$

$$\text{(T-JSSupSelect)}$$

$$\dfrac{\Gamma \vdash \texttt{t0:S0} \quad \texttt{S0 <: any} \quad \Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: any} \quad \Gamma \vdash \texttt{t3:S3} \quad \texttt{S3 <: any}}{\Gamma \vdash \texttt{super(t0)::t1[t2] = t3 : void}}$$

(T-JSAssSupSelect)

$$\dfrac{\Gamma \vdash \texttt{t0:S0} \quad \texttt{S0 <: any} \quad \Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: any} \quad \Gamma \vdash \overline{\texttt{t}}:\overline{\texttt{S}} \quad \overline{\texttt{S}} \texttt{ <: any}}{\Gamma \vdash \texttt{super(t0)::t1[t2]}(\overline{\texttt{[...]t}}) \texttt{ : any}}$$

(T-JSSupMethApply)

$$\dfrac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any}}{\Gamma \vdash \texttt{(+|-|~|!)t1 : any}}$$

(T-JSUnOp)

$$\dfrac{\Gamma \vdash \texttt{t1:S1} \quad \texttt{S1 <: any} \quad \Gamma \vdash \texttt{t2:S2} \quad \texttt{S2 <: any}}{\Gamma \vdash \texttt{t1 (+|-|*|/|\%|||\&|^|<<|>>|>>>|<|<=|>|>=|||\&\&}}$$
$$\texttt{|in|instanceof) t2 : any}$$

(T-JSBinOp)

$$\dfrac{\texttt{clskind(C)} \in \{\texttt{js class,js module class,native js class}\} \quad \texttt{istoplevel(C)}}{\Gamma \vdash \texttt{constructorOf[C] : any}}$$

(T-LoadJSCtor)

$$\dfrac{\texttt{clskind(C)} \in \{\texttt{js module class,native js module class}\}}{\Gamma \vdash \texttt{mod:C : any}}$$

(T-LoadJSModule)

$$\dfrac{\Gamma \vdash \overline{\texttt{t}}:\overline{\texttt{S}} \quad \overline{\texttt{S}} \texttt{ <: any}}{\Gamma \vdash \texttt{[}\overline{\texttt{[...]t}}\texttt{] : any}}$$

(T-JSArray)

$$\dfrac{\Gamma \vdash \overline{\texttt{t1}}:\overline{\texttt{S1}} \quad \overline{\texttt{S1}} \texttt{ <: any} \quad \Gamma \vdash \overline{\texttt{t2}}:\overline{\texttt{S2}} \quad \overline{\texttt{S2}} \texttt{ <: any}}{\Gamma \vdash \texttt{\{}\overline{\texttt{[t1]: t2}}\texttt{\} : any}}$$

(T-JSObject)

$$\dfrac{\texttt{x is a valid JavaScript identifier}}{\Gamma \vdash \texttt{global:x : any}}$$

(T-JSGlobalRef)

Literals

$$\Gamma \vdash \texttt{null : null} \quad \text{(T-NullLit)} \qquad \Gamma \vdash \texttt{<short literal> : short}$$

(T-ShortLit)

$$\Gamma \vdash \texttt{<boolean literal> : boolean}$$

(T-BooleanLit)

$$\Gamma \vdash \texttt{<int literal> : int} \quad \text{(T-IntLit)}$$

$$\Gamma \vdash \texttt{<char literal> : char} \quad \text{(T-CharLit)}$$

$$\Gamma \vdash \texttt{<byte literal> : byte} \quad \text{(T-ByteLit)} \qquad \Gamma \vdash \texttt{<long literal> : long} \quad \text{(T-LongLit)}$$

$$\Gamma \vdash \texttt{<float literal> : float} \qquad\qquad \Gamma \vdash \texttt{<string literal> : string}$$
$$\text{(T-FLOATLIT)} \qquad\qquad\qquad\qquad\qquad \text{(T-STRINGLIT)}$$

$$\Gamma \vdash \texttt{<double literal> : double}$$
$$\text{(T-DOUBLELIT)} \qquad \Gamma \vdash \texttt{undefined : undef} \;\text{(T-UNDEFLIT)}$$

Closures and class creation

$$\frac{\overline{\texttt{T1}} \neq \texttt{void} \quad \Gamma \vdash \overline{\texttt{t1}:\texttt{S1}} \quad \overline{\texttt{S1} \; \texttt{<:} \; \overline{\texttt{T1}}} \quad \varnothing, \overline{\texttt{x1}:\texttt{T1}}, \overline{\texttt{x2}:\texttt{any}} \vdash \texttt{t}:\texttt{any}}{\Gamma \vdash \texttt{arrow-lambda<}\overline{\texttt{x1}:\texttt{T1=t1}}\texttt{>(}[\texttt{var}] \; [\texttt{...}] \; \overline{\texttt{x2}:\; \texttt{any}}\texttt{) = t : any}} \; \text{(T-ARROWLAMBDA)}$$

$$\frac{\overline{\texttt{T1}} \neq \texttt{void} \quad \Gamma \vdash \overline{\texttt{t1}:\texttt{S1}} \quad \overline{\texttt{S1} \; \texttt{<:} \; \overline{\texttt{T1}}} \quad \varnothing, \texttt{this:any}, \overline{\texttt{x1}:\texttt{T1}}, \overline{[\texttt{var}]\texttt{x2}:\texttt{any}} \vdash \texttt{t}:\texttt{any}}{\Gamma \vdash \texttt{function-lambda<}\overline{\texttt{x1}:\texttt{T1=t1}}\texttt{>(}[\texttt{var}] \; [\texttt{...}] \; \overline{\texttt{x2}:\; \texttt{any}}\texttt{) = t : any}}$$
$$\text{(T-FUNCTIONLAMBDA)}$$

$$\frac{\overline{\texttt{x1}:\texttt{T1}} \; \texttt{js class C [extends C1 [via t]] [implements } \overline{\texttt{C2}} \texttt{] \{ } \overline{\texttt{CM}} \texttt{ \}} \quad \Gamma \vdash \overline{\texttt{t1}:\texttt{S1}} \quad \overline{\texttt{S1} \; \texttt{<:} \; \overline{\texttt{T1}}}}{\Gamma \vdash \texttt{createjsclass[C](}\overline{\texttt{t1}}\texttt{) : any}}$$
$$\text{(T-CREATEJSCLASS)}$$

Specials

$$\Gamma \vdash \texttt{<linking-info>: any} \qquad\qquad \text{(T-LINKINGINFO)}$$

Appendix B

RuntimeLong annotated with Predicate-Qualified Types

```scala
class RuntimeLong(val lo: Int, val hi: Int) {
  import Utils._

  val toLong: Long =
    (this.lo.toLong & 0xffffffffL) | (this.hi.toLong << 32L)


  // Comparators

  def <(b: RuntimeLong): {v: Boolean =>
      v == this.toLong < b.toLong} = {
    if (this.hi == b.hi) (this.lo ^ 0x80000000) < (b.lo ^ 0x80000000)
    else this.hi < b.hi
  }

  def <=(b: RuntimeLong): {v: Boolean =>
      v == this.toLong <= b.toLong} = {
    if (this.hi == b.hi) (this.lo ^ 0x80000000) <= (b.lo ^ 0x80000000)
    else this.hi < b.hi
  }

  def >(b: RuntimeLong): {v: Boolean =>
      v == this.toLong > b.toLong} = {
    if (this.hi == b.hi) (this.lo ^ 0x80000000) > (b.lo ^ 0x80000000)
    else this.hi > b.hi
  }

  def >=(b: RuntimeLong): {v: Boolean =>
      v == this.toLong >= b.toLong} = {
    if (this.hi == b.hi) (this.lo ^ 0x80000000) >= (b.lo ^ 0x80000000)
    else this.hi > b.hi
  }
```

```scala
def equals(c: RuntimeLong): {v: Boolean =>
    v == (this.toLong == c.toLong)} =
  (this.lo == c.lo) && (this.hi == c.hi)


def notEquals(b: RuntimeLong): {v: Boolean =>
    v == (this.toLong != b.toLong)} = {
  val eq = this.equals(b): {v: Boolean =>
      v == (this.toLong == b.toLong)}
  (!eq): {v: Boolean => v == (this.toLong != b.toLong)}
}



// Bitwise operations

def unary_~ : {v: RuntimeLong => v.toLong == ~(this.toLong)} = {
  new RuntimeLong(~this.lo, ~this.hi)
}


def |(b: RuntimeLong): {v: RuntimeLong =>
    v.toLong == (this.toLong | b.toLong)} = {
  new RuntimeLong(this.lo | b.lo, this.hi | b.hi)
}


def &(b: RuntimeLong): {v: RuntimeLong =>
    v.toLong == (this.toLong & b.toLong)} = {
  new RuntimeLong(this.lo & b.lo, this.hi & b.hi)
}


def ^(b: RuntimeLong): {v: RuntimeLong =>
    v.toLong == (this.toLong ^ b.toLong)} = {
  new RuntimeLong(this.lo ^ b.lo, this.hi ^ b.hi)
}



// Shifts

def <<(n: Int) = {
  val lo = if ((n & 32) == 0) this.lo << n else 0
  val hi =
    if ((n & 32) == 0) (this.lo >>> 1 >>> (31-n)) | (this.hi << n)
    else this.lo << n
  new RuntimeLong(lo, hi): {v: RuntimeLong =>
      v.toLong == (this.toLong << n.toLong)}
}

def >>>(n: Int) = {
  val lo =
    if ((n & 32) == 0) (this.lo >>> n) | (this.hi << 1 << (31-n))
    else this.hi >>> n
  val hi = if ((n & 32) == 0) this.hi >>> n else 0
  new RuntimeLong(lo, hi): {v: RuntimeLong =>
      v.toLong == (this.toLong >>> n.toLong)}
}
```

```scala
  def >>(n: Int) = {
    val lo =
      if ((n & 32) == 0) (this.lo >>> n) | (this.hi << 1 << (31-n))
      else this.hi >> n
    val hi = if ((n & 32) == 0) this.hi >> n else this.hi >> 31
    new RuntimeLong(lo, hi): {v: RuntimeLong =>
        v.toLong == (this.toLong >> n.toLong)}
  }


  // Arithmetic

  def unary_- = {
    val hi = if (this.lo != 0) ~this.hi else -this.hi
    new RuntimeLong(-this.lo, hi): {v: RuntimeLong =>
        v.toLong == -this.toLong}
  }

  def +(b: RuntimeLong) = {
    val lo = this.lo + b.lo
    val overflowed = inlineUnsignedInt_<(lo, this.lo)
    val hi = if (overflowed) this.hi + b.hi + 1 else this.hi + b.hi
    new RuntimeLong(lo, hi): {v: RuntimeLong =>
        v.toLong == (this.toLong + b.toLong)}
  }

  def -(b: RuntimeLong) = {
    val lo = this.lo - b.lo
    val underflowed = inlineUnsignedInt_>(lo, this.lo)
    val hi = if (underflowed) this.hi - b.hi - 1 else this.hi - b.hi
    new RuntimeLong(lo, hi): {v: RuntimeLong =>
        v.toLong == (this.toLong - b.toLong)}
  }


  private object Utils {
    def inlineUnsignedInt_<(a: Int, b: Int): Boolean =
      (a ^ 0x80000000) < (b ^ 0x80000000)

    def inlineUnsignedInt_>(a: Int, b: Int): Boolean =
      (a ^ 0x80000000) > (b ^ 0x80000000)
  }
}

object RuntimeLong {
  def apply(lo: Int, hi: Int) =
    new RuntimeLong(lo, hi)

  def apply(lo: Int): {v: RuntimeLong => v.toLong == lo.toLong} =
    new RuntimeLong(lo, lo >> 31)
}
```

Bibliography

[1] K. Ali. *The Separate Compilation Assumption*. PhD thesis, University of Waterloo, 2014.

[2] K. Ali and O. Lhoták. Application-only call graph construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 688–712, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] asm.js. http://asmjs.org/, 2016.

[4] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, Lecture Notes in Computer Science, pages 171–177. Springer, July 2011. Snowbird, Utah.

[5] M. Bodin, A. Chargueraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 87–100, 2014.

[6] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Trans. Program. Lang. Syst.*, 15(3):367–399, July 1993.

[7] E. Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013.

[8] C. Chambers, J. Dean, and D. Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 221–230, New York, NY, USA, 1995.

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

[10] CoffeeScript. http://coffeescript.org/, 2015.

[11] K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 58–67, New York, NY, USA, 1986.

Bibliography

[12] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] J. A. Dean. *Whole-program Optimization of Object-oriented Languages*. PhD thesis, University of Washington, 1996. AAI9716832.

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

[15] S. Doeraene and T. Schlatter. Parallel incremental whole-program optimizations for Scala.js. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 59–73, 2016.

[16] S. Doeraene, T. Schlatter, and N. Stucki. Scala.js. https://www.scala-js.org/, 2013–2018.

[17] S. Doeraene, T. Schlatter, and N. Stucki. Semantics-driven interoperability between Scala.js and JavaScript. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, SCALA 2016, pages 85–94, 2016.

[18] Ecma International. *ECMAScript 2015 Language Specification*. Ecma International, Geneva, 6th edition, June 2015.

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

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

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

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

[23] Google. GWT Long implementation. https://github.com/gwtproject/gwt/tree/2.7.0/dev/core/super/com/google/gwt/lang, 2014.

[24] Google. Closure Library. https://developers.google.com/closure/library/, 2015.

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

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

[27] Graal.js Contributors. Graal.js. https://github.com/graalvm/graaljs, 2018.

[28] T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 61–72, 1994.

[29] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, 2017.

[30] Haxe Foundation. Haxe. http://haxe.org/, 2015.

[31] R. Hyde. *The Art of Assembly Language*. No Starch Press, 2nd edition, 2010.

[32] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[33] JetBrains. Kotlin. https://kotlinlang.org/, 2016.

[34] JetBrains. Kotlin Long implementation. https://github.com/JetBrains/kotlin/blob/v1.1. 4-3/js/js.libraries/src/js/long.js, 2017.

[35] I. Lakhin. Papa Carlo, 2015. [Online; accessed 30-November-2015].

[36] D. Leopoldseder, L. Stadler, C. Wimmer, and H. Mössenböck. Java-to-JavaScript translation via structured control flow reconstruction of compiler IR. In *Proceedings of the 11th Symposium on Dynamic Languages*, DLS 2015, pages 91–103, 2015.

[37] H. Li. Autowire, 2015. [Online; accessed 30-November-2015].

[38] L. Liquori and A. Spiwack. Extending FeatherTrait Java with Interfaces. *Theoretical Computer Science*, 30(1-3):243–260, May 2010.

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

[40] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 61–81, Berlin, Heidelberg, 2013. Springer-Verlag.

[41] S. Marr, B. Daloze, and H. Mössenböck. Cross-language compiler benchmarking: Are we fast yet? In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 120–131, 2016.

[42] 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.

[43] mbedTLS. https://github.com/ARMmbed/mbedtls, 2015.

[44] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10:100–107, 1998.

Bibliography

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

[46] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 183–202, New York, NY, USA, 2013.

[47] Octane, the JavaScript benchmark suite for the modern web, 2015. [Online; accessed 30-November-2015].

[48] M. Odersky and Contributors. Scala language specification. https://www.scala-lang.org/files/archive/spec/2.12/, 2017.

[49] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[50] L. L. Pollock and M. L. Soffa. Incremental compilation of optimized code. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 152–164, New York, NY, USA, 1985.

[51] L. L. Pollock and M. L. Soffa. Incremental global reoptimization of programs. *ACM Trans. Program. Lang. Syst.*, 14(2):173–200, Apr. 1992.

[52] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 151–160, New York, NY, USA, 2012.

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

[54] Red Hat. Ceylon. https://ceylon-lang.org/, 2016.

[55] D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 117–128, New York, NY, USA, 2005.

[56] G. S. Schmid and V. Kuncak. SMT-based checking of predicate-qualified types for Scala. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, SCALA 2016, pages 31–40, 2016.

[57] A. Sewe, J. Jochem, and M. Mezini. Next in line, please!: Exploiting the indirect benefits of inlining by accurately predicting further inlining. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 317–328, New York, NY, USA, 2011.

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

[59] A. L. Souter and L. L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 682–, Washington, DC, USA, 2001.

[60] L. Stadler, T. Würthinger, and H. Mössenböck. Partial escape analysis and scalar replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014.

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

[62] TeaVM. http://teavm.org/, 2015.

[63] TeaVM Long implementation. https://github.com/konsoletyper/teavm/blob/0.5.1/core/src/main/resources/org/teavm/backend/javascript/runtime.js#L623, 2016.

[64] The Clang Team. UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2017–2018.

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

[66] J. Vilk and E. D. Berger. Doppio: Breaking the browser language barrier. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 508–518, 2014.

[67] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 35–46, New York, NY, USA, 2001.

[68] S. Waser and M. J. Flynn. *Introduction to Arithmetic for Digital Systems Designers.* Harcourt Brace College Publishers, 1995.

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

[70] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676, 2017.

[71] J.-S. Yur, B. G. Ryder, W. A. Landi, and P. Stocks. Incremental analysis of side effects for C software system. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 422–432, New York, NY, USA, 1997.

Bibliography

[72] A. Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 301–312, 2011.

Curriculum Vitae

Sébastien Doeraene

Chemin de la Chiésaz 7
1024 Écublens VD
Switzerland
✆ +41 78 658 85 32
✉ sjrdoeraene@gmail.com
🖥 sebastien.doeraene.be

Education

2013–today   **PhD**, *École polytechnique fédérale de Lausanne (ongoing)*.
PhD thesis at the Programming Methods Laboratory (LAMP) under the supervision of Prof. Martin Odersky.
My research is focused on the design and development of the Scala.js programming language dialect (`https://www.scala-js.org/`), featuring portability with respect to Scala/JVM, interoperability with JavaScript, and competitive performance.

2006–2011   **MSc Computer Science and Engineering**, *Université catholique de Louvain (Louvain School of Engineering)*, *high honors*.
Option: networks and security
Minor option: linguistics (Faculty of Philosophy, Arts and Letters)

Master's thesis

title   *Ozma: Extending Scala with Oz Concurrency*

supervisor   Prof. Peter Van Roy

description   Design and implementation of a dialect of Scala with declarative concurrency, targeting the Mozart/Oz virtual machine.

awards   Best master's thesis in computer science at UCL in 2011 (1st out of 40 theses).

Awards

2010   Prologin, national French programming contest. First prize.
`https://prologin.org/`

Teaching Experience

2013–2017   **Teaching Assistant**, *École polytechnique fédérale de Lausanne*.
Supervision of weekly exercise sessions, design and grading of exams. Head TA of CS-210 Functional programming.

2011–2013   **Teaching Assistant**, *Université catholique de Louvain*.
Supervision of weekly exercise sessions, design and grading of exams and term projects.

Selected Publications

Scala'16 *Sébastien Doeraene*, Tobias Schlatter, and Nicolas Stucki. Semantics-Driven Interoperability between Scala.js and JavaScript. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, 2016.

OOPSLA'16 *Sébastien Doeraene* and Tobias Schlatter. Parallel Incremental Whole-Program Optimizations for Scala.js. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.

Scala'13 *Sébastien Doeraene* and Peter Van Roy. A New Concurrency Model for Scala based on a Declarative Dataflow Core. In *Proceedings of the 4th Workshop on Scala*, 2013.

Experience in Language and Compiler Design

2013–today **Lead designer of Scala.js**, *École polytechnique fédérale de Lausanne*.
Design of the Scala.js dialect, development of the compiler and standard library.
`https://www.scala-js.org/`

2011–2013 **Co-lead designer of Mozart 2**, *Université catholique de Louvain*.
Development of the Mozart Virtual Machine and design of the reflective subsystem of Oz.
`https://mozart.github.io/`

Languages

French	Native speaker	
English	Proficient	*scored 8/9 at the IELTS in 2009, equivalent to CEFR level C2*
Spanish	Intermediate	*self-assessed CEFR level B1*
Dutch	Elementary	*self-assessed CEFR level A2*
Japanese	Notions	*self-assessed JLPT level N5*
German	Notions	*elementary reading comprehension*

Miscellaneous

music degree	Music theory, Transition	*Conservatoire de Musique de Tournai, Belgium, 2003*
singing	Choir and a cappella singing, with a soft spot for movie soundtracks	
sports	Unicycle and spikeball	
volunteering	President of the student association Ensemble Vocal Évohé	
driving	B driving license	