SJS-JFE: A Java Frontend for Scala.js

Martin Hanzel
313710

Supervised by
Sébastien Doeraene

Submitted
June 2020

Abstract

This report presents SJS-JFE, a Java compiler that targets the Scala.js IR. Java programs compiled with SJS-JFE can run on the Scala.js platform and link with other Scala.js code written in either Java or Scala. This interoperability opens up the possibility of using Java libraries in Scala.js projects, greatly widening the Scala.js ecosystem.

Contents

1 Introduction

Scala.js[1] is a dialect and compiler for the Scala language that targets JavaScript platforms. Scala.js can compile a Scala program into an optimized JavaScript program that can be run on server-side JS engines (like Node.js), as well as browsers. It also provides bindings to JavaScript APIs so that a Scala.js program can interact with the browser or the runtime as a first-class JavaScript program could.

Scala.js applications can take advantage of Scala language features — for instance, strong typing, correctness guarantees, and a comprehensive standard library — which vanilla Javascript lacks. This gap is being rapidly closed by alternative languages such as Typescript[2], though Scala arguably remains more powerful, with a better type system and more language features than competing languages.

Scala.js is introduced and described in Sébastien Doeraene's PhD thesis [1]. The reader of this report is assumed to have at least a basic knowledge of Scala.js and its IR.

1.1 Motivation

Scala.js programs are written in Scala, and can interoperate with JavaScript code and a subset of the Scala standard library. A subset of the Java standard library is also supported, having been re-implemented in Scala. However, there is no facility to call Java code (either from source or from bytecode), meaning that a Scala.js application cannot exploit the extensive Java ecosystem.

This void is filled by SJS-JFE (**S**cala.**js**-**J**ava **F**ront**e**nd, for lack of a better name), a standalone compiler based on the Eclipse Java Development Tools (JDT)[3]. SJS-JFE transforms Java source code to the Scala.js intermediate language so that it may be linked with other Scala.js code and finally transformed into JavaScript.

SJS-JFE receives as input a Java source file, which is processed in two stages. In the first stage, SJS-JFE invokes the JDT compiler, which parses the source, checks the code for errors, and finally generates one or more ASTs representing all top-level declarations in the source file. In the second stage, SJS-JFE transpiles a JDT AST into Scala.js IR. This IR, stored on disk in files with the `.sjsir` extension, is somewhat an analogue of bytecode in the JVM world.

After an IR file is generated, it can be linked with other IR files using the Scala.js compiler to assemble more complex programs. Notably, it is possible to link IR files compiled from Java sources with ones compiled from Scala sources. The major consequence of this is that Scala programs are able to utilize Java libraries by compiling the library's sources to Scala.js IR, greatly increasing the size of the Scala.js ecosystem.

SJS-JFE implements a one-pass, recursive transpiler. Its job is greatly simplified by the fact that the Scala.js IR class model is quite similar to Java's. Excepting some structural changes to the top-level declarations, much of Scala.js IR code mirrors its Java equivalent quite closely.

[1] http://www.scala-js.org/
[2] https://www.typescriptlang.org/
[3] https://www.eclipse.org/jdt/

SJS-JFE assumes that the source code it receives as input is correct — if it is not, the JDT compiler will detect and indicate errors in the first stage, and fail early. Thanks to this precaution, the second stage transpiler does not need to deal with any error-checking, and focus on emulating Java's semantics.

This report frequently references the Java Language Specification, version 14 [2]. The curious reader is encouraged to have it at hand to better understand the details of Java semantics.

The source code to SJS-JFE is available on GitHub[4].

1.2 Example Program

An example is worth a thousand words. Consider the following "Hello World" program in Java:

```
class Main {
  String name = "anonymous";
  public Main(String name) {
    this.name = name;
  }
  public void print(String[] args) {
    System.out.println(name + " says:");
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
  public static void main() {
    new Main().print(new String[] { "Hello", "Scala.js" });
  }
}
```

SJS-JFE compiles it to this Scala.js IR:

```
class Main extends java.lang.Object {
  var name: java.lang.String
  def print;[Ljava.lang.String;V(args: java.lang.String[]) {
    _return_0: {
      java.lang.System::out;Ljava.io.PrintStream()
        .println;Ljava.lang.String;V(
          (this.Main::name +[string] " says:")
        );
      var i: int = 0;
      _break_0: {
        while ((i <[int] args.length)) {
          _continue_0: {
            java.lang.System::out;Ljava.io.PrintStream()
              .println;Ljava.lang.String;V(args[i])
          };
          i = (i +[int] 1)
        }
      }
    }
  }
  static def main;V() {
    _return_1: {
```

[4] https://github.com/arthanzel/scalajs-jfe

```
      new Main().<init>;Ljava.lang.String;V("SJS-JFE")
        .print;[Ljava.lang.String;V(
          java.lang.String[]("Hello", "Scala.js")
        )
    }
  }
  constructor def <init>;Ljava.lang.String;V(
      name: java.lang.String) {
    this.java.lang.Object::<init>;V();
    this.Main::name = "anonymous";
    this.Main::name = name
  }
}
```

Some lines were wrapped to fit the code on the page.

The IR representation is definitely more verbose, but from this example, we can note a few things right away:

- Methods, including constructors, declare their full signature, including parameter and return types, in the name. This is a feature of Scala.js that allows method overloading in JavaScript.
- The `for` loop is converted into a `while`.
- Constructors explicitly call the default super-constructor.
- Instance fields are initialized in the constructor, instead of in their declaration.

Chapter 2 describes all of these transformations, and more, that are applied when converting Java source code to Scala.js IR.

1.3 Design Decisions

We considered two possibilities: building a compiler that takes Java source code as input, or one that takes JVM bytecode. Both options have benefits and drawbacks:

- A bytecode frontend could feasibly support not only Java, but other languages compiling to JVM bytecode as well: for example Kotlin, and Clojure. JVM bytecode is quite primitive, and eliminates tricky constructs that can appear in source code, like nested classes, closures, and any syntactic-sugar constructs such as `for-in` loops. On the other hand, the low-level nature of bytecode makes it difficult to analyze. Generic type information is lost due to type erasure. Further, the JVM bytecode is much more low-level than the Scala.js IR and there is not necessarily a clean way to map bytecode instructions to Scala.js — any attempt to transform bytecode to Scala.js IR runs the risk of producing Yet Another JVM Implementation.
- A source code frontend is limited to consuming Java programs, but its task would be much simpler. The Scala.js IR is conceptually similar to the Java language model, and many concepts have a one-to-one counterpart. Full type information is retained, including erasures, parameterized type

bounds, and original types of erasures at the site of usage. Language-level optimizations may be performed. Static analysis and data flow analysis can be applied. If a sufficiently-advanced Java parser is used, transforming Java source to the Scala.js IR becomes a matter of transforming one structured AST into another. If the source code is not available, JVM bytecode may be first decompiled into Java and then transformed into the Scala.js IR (assuming that the proper libraries are present and that the bytecode is compliant to the JVM specification).

SJS-JFE takes the latter option — it consumes Java source code. This path was chosen as it is the simpler of the two, and naturally, the path of least resistance is desirable for writing a proof-of-concept program. In the future, we'd like to explore the possibility of a bytecode compiler as well.

Recall, the first stage of SJS-JFE is to parse the Java source into an AST and check it for errors. The Eclipse JDT[5] suite was chosen as the first-stage compiler since it is a mature, full-featured, and battle-tested compiler suite. Its benefits are numerous:

- JDT is available under the permissive Eclipse Public License, unlike the OpenJDK project, which is licensed under the GPL. SJS-JFE merely interfaces with JDT and is not a derivative work, making JDT compatible with the Apache 2.0 license of Scala.js.
- JDT is the default compiler and toolchain for Java projects in the Eclipse IDE, giving it an excellent track record.
- JDT has flexible configuration and exposes its internals. It can return fully-generated or partially-generated ASTs. In contrast, the `javac` API is opaque, poorly-documented, and less full-featured.
- JDT resolves type/method/variable bindings and provides comprehensive information about types, including erasures for parameterized types.

1.4 Limitations and Challenges

Language features The current state of SJS-JFE supports the majority of Java's language features — certainly enough to write reasonably complex programs. Some features are not yet implemented or well-tested, chiefly: `try-catch` blocks, lambda expressions, anonymous classes, local classes, `switch` expressions, and enhanced `for` loops.

Implementing the remaining Java language features is a priority, but Java is a complex language with many language features that interact in non-trivial ways. For example, a concept as fundamental as implicit value conversions (JLS14 §5) has many pages of rules, preconditions, and contexts dictating where it can and cannot be applied. The PDF of the Java 6 Language Specification[6] comprises nearly 700 pages. The PDF of the Java 14 Language Specification[7] has nearly 800!

[5] SJS-JFE uses JDT version 3.21.0

[6] https://docs.oracle.com/javase/specs/jls/se6/jls3.pdf

[7] https://docs.oracle.com/javase/specs/jls/se14/jls14.pdf

Java API Scala.js supports only a subset of the Java standard library, which is re-implemented from scratch[8]. Since Scala.js code is eventually run in a JavaScript environment (including browsers), JVM-specific or server-specific functionality is unsupported. Among these missing features are reflection, file-system operations, compile-time annotations, and concurrency, for example.

Naturally, any Java standard library functionality that cannot be emulated using the JavaScript standard library is unsupported.

DOM API Scala.js allows Scala code to interface with JavaScript APIs, including the JS standard library and the DOM API on browsers, through a set of native bindings. SJS-JFE provides no similar functionality at this time, so Java code cannot access JS APIs. Consequently, SJS-JFE does not yet enable Java to be used as the primary language for developing useful, real-world Scala.js programs.

[8] see the `javalib` and `javalanglib` modules in the Scala.js main repository

2 Translation

This section describes the various rules that are applied to transform a JDT AST to the Scala.js IR. A thorough introduction to the Scala.js IR is not included; for that, refer to the original thesis [1], or peruse the Scala.js source code[9]. The differences between Java and Scala.js semantics will be explained wherever relevant, but the reader should not take this as a guide to Scala.js — rather, as a manual of how the JDT model maps to Scala.js. Consequently, a basic understanding of the Scala.js IR is assumed, or at least a decent knowledge of ASTs and tree-based compilers. Several code examples include listings of pretty-printed Scala.js IR. The reader is assumed to be familiar with reading Scala.js IR code.

JDT AST classes live in the `org.eclipse.jdt.core.dom` package. Throughout this report, JDT classes may be referred to using the shorthand `jdt.<ClassName>`.

There are three main types of translation rules that SJS-JFE applies to a JDT AST to transform it into Scala.js IR:

1. **High-level structure rules.** These rules deal with assembling class definitions and top-level members like methods, fields, and static initializers. High-level structure rules are the most complicated, since they must adapt the public interface of a Java class to match Scala.js semantics. Among some problems that these rules must solve are generation of implicit constructors, synthetic accessors, static initialization, and handling of lexically enclosing scopes.

2. **Statement rules.** These rules correspond to Java statements. Statements may appear only within initializers or methods, and do not affect the overall structure of the class. Statements may wrap expressions.

3. **Expression rules.** These rules correspond to Java expressions. Expressions are treated as a special case because they may contain additional expressions and may even appear in a statement position.

A remark on notation: Throughout this chapter, listings are provided to summarize the transformation rules that SJS-JFE applies to a JDT AST to produce Scala.js IR. They take the following form:

```
[[ javaPattern ]] := scalajsPattern
```

Elements inside `[[` double square brackets `]]` are transformed by SJS-JFE into Scala.js IR, that is, `[[ X ]]` maps a JDT tree `X` to a Scala.js IR. On the left-hand side of the `:=` appear pseudocode representations of the JDT AST. On the right-hand side are pseudocode representations of the Scala.js IR. They are not equivalent. For example, consider the following transformation:

```
[[ if (cond) { thenP } else { elseP } ]] :=
  if ([[ cond ]]) { [[ thenP ]] } else { [[ elseP ]] }
```

[9] https://github.com/scala-js/scala-js — the `org.Scala.js.ir.Trees` object contains all nodes in the IR

This encodes a very basic transformation from a Java `if-else` statement to the corresponding `if-else` node in the Scala.js IR. The `if` on the left-hand side corresponds to a JDT `if` node, while the one on the right corresponds to its Scala.js IR counterpart.

For the most part, Scala.js IR is shown as it would be printed by the compiler. However, sometimes certain details may be omitted (like type information in method names) for the sake of brevity.

2.1 High-level Structure Rules

When JDT parses a Java source file, it returns an instance of `jdt.CompilationUnit` containing zero or more instances of `jdt.AbstractTypeDeclaration`. An `AbstractTypeDeclaration` can declare an annotation, enum, interface, or class.

- Scala.js does not support annotations, so SJS-JFE ignores annotation declarations entirely.
- Enums are not supported yet by SJS-JFE.
- Classes and interfaces are both instances of `jdt.TypeDeclaration`, and are processed in a similar fashion. Interfaces are not yet tested, however, they are treated in the same way as classes. Empty interface methods and abstract methods are defined as regular methods with an undefined body.

A source file may define many types, either at the top-level or nested within another type. Each type declaration is treated and emitted individually, similar to how all type declarations, nested or otherwise, appear in their own `.class` files.

2.1.1 Class Generation

When SJS-JFE encounters a class definition, it performs the following transformations:

- If the class does not explicitly name a superclass, set the superclass to `java.lang.Object`.
- If the class does not declare any constructor, create a default zero-arg constructor.
- If the class does declare constructors, add to them synthetic code to initialize the class, its instance members, and any statics.
- If the class is an inner class, add a field holding a reference to the lexically enclosing instance and amend constructors to receive and set this reference.
- Process and generate fields. If the class declares static fields, generate synthetic getters and setters.
- If the class declares static fields or static initializers, generate a synthetic method to carry out static initializations in order.

- Process and generate methods.
- If the class declares inner types, process them recursively.
- The name of the class is set to its JVM binary name.

SJS-JFE returns a collection of Scala.js `ClassDef` instances for the root class and any inner class.

2.1.2 Instance Fields

A class may declare zero or more instance fields as members. One significant difference between the Java and Scala.js models is that Java fields are allowed to include a value to use as an initializer, whereas Scala.js does not. In Scala.js, instance fields must be set to their proper values in constructors.

Another difference is that a Java field declaration may declare multiple variables ("fragments") of the same type. Each fragment may have a different initial value, for example, `int a = 1, b = 2;`. Scala.js requires that each fragment become its own field declaration.

When SJS-JFE encounters a Java field declaration, it splits each fragment into its own declaration and constructs a Scala.js `FieldDef` node. Initializers are emitted during constructor generation.

2.1.3 Constructors

Java implicitly creates a blank, zero-arg constructor if a class does not declare any constructors *and* the superclass is implicitly constructable (i.e. it either has an implicit constructor or an explicit zero-arg constructor). Scala.js requires that all constructors be explicit, therefore SJS-JFE generates a default constructor if needed.

Default constructor: The default constructor is quite simple. It performs only four major tasks:

1. If the class is a nested class, store a reference to the enclosing instance in a synthetic field from a parameter.
2. Call the default super constructor. This call is necessary to ensure that any fields from superclasses are initialized.
3. If the class declares static fields or initializers, call the static initialization routine (see section 2.1.4, and the note below).
4. Set instance fields to their initial values.

In fact, generating a default constructor in the IR is equivalent to inserting a blank zero-arg constructor in the JDT AST, and processing it as an explicit constructor. However, it is inconvenient to create synthetic AST elements with the proper binding information, so implicit constructors are treated separately.

Note: The current state of SJS-JFE does not perform static initialization in the correct order with respect to super-constructor calls. The correct behaviour is for all classes and subclasses to be statically initialized, starting from the

base class, before *any* constructor executes. In the current implementation, static initializers are called alongside their class's constructor, starting from the base class. Normally, this is not an issue unless the classes have static initializers and constructors with side-effects that are sensitive to their order of execution. The reason for this is that classes cannot know whether their superclass has a static initialization routine, so the only way for a class's static initializer to be called reliably is from the class itself — for example from the constructor.

Explicit constructors. If a class contains explicit constructors, no default constructor is created and all explicit constructors are augmented by prepending some synthetic code to the body, if necessary:

1. If the class is a nested class, store a reference to the enclosing instance in a synthetic field from a parameter.
2. If the explicit constructor explicitly calls a super constructor and the superclass is a nested class, amend the super constructor call with a reference to the appropriate enclosing instance.
3. If the explicit constructor call no other constructor, generate a call to the default super constructor, passing a reference to the appropriate enclosing instance if applicable.
4. If the class declares static fields or initializers, call the static initialization routine (see section 2.1.4).
5. If the constructor is a top-level constructor, set instance fields to their initial values.

One difference between Java and Scala is worth noting here. In Scala, classes are guaranteed to have a single primary constructor. That is, there must exist one constructor that is called no matter how the class is instantiated — all auxiliary constructors must either call the primary or another auxiliary constructor. In Java, there may exist several "top-level" constructors, which either call a super-constructor as their first statement, or call no constructor explicitly (therefore implicitly invoking the zero-arg super constructor). Initialization code, like the setting of instance fields, must be duplicated in all top-level constructors so that it is guaranteed to be run. A future optimization may be to extract this initialization code into a synthetic method, if the goal is to minimize the size of the IR.

2.1.4 Statics

A major difference between Java and Scala is their representation of statics. In Scala, "static" members are declared in a *companion object*, which is effectively a singleton containing instance members. Indeed, in Scala.js, companion objects are compiled into *modules*, a kind of object that wraps a singleton. However, for interoperability with Java, the Scala.js IR has support for static members. Static fields and methods can be declared with a "static" flag, and are accessed with specialized AST nodes. The *declaration* of static members is not an issue in Scala.js.

Recall from section 2.1.2 that Scala.js field declarations may not set their initial value — this is done in the constructor. Scala.js modules do support a constructor-like method that instantiates the singleton and sets initial values. But there is no "static constructor" equivalent for classes. So where are static fields supposed to be initialized?

A related concern is that of static initializers. A Java class may declare zero or more static initializers, contained within `static { }` blocks, that perform one-time tasks whenever (roughly) the class is first referenced. Static initializers may overwrite fields or perform other (expensive) side-effects, so calling them at the right time is important to preserve Java semantics. Therefore, doing static initialization for all classes when the program is loaded would be naive and wasteful.

A class is initialized immediately before it is instantiated, one of its static methods is invoked, one of its static fields is written, one of its non-final static fields is read, or a subclass is initialized (JLS14 §12.4.1). SJS-JFE hooks onto these events by creating synthetic accessors for each static field, where the accessors call a synthetic initializer method. The synthetic initializer is guarded by another synthetic field to ensure that the method is called at most once. The role of the synthetic initializer is to perform all static initialization code and set all static variables, in the order in which they appear in the source.

If a static field is read or written elsewhere in the code, it is accessed through the appropriate getter or setter, respectively.

Consider the following class:

```
class MyClass {
    static int test = 10;
    static { System.out.println("init"); }
}
```

This is the IR that SJS-JFE would emit:

```
class Main extends java.lang.Object {
  static var test: int
  static def test;I(): int = {              // Getter
    Main::$sjsirStaticInitializer_0;V();
    Main::test
  }
  static def test_$eq;I;V(value: int) {     // Setter
    Main::$sjsirStaticInitializer_0;V();
    Main::test = value
  }
  static var $sjsirStaticCalled_0: boolean
  static def $sjsirStaticInitializer_0;V() {
    if ((!Main::$sjsirStaticCalled_0)) {
      Main::$sjsirStaticCalled_0 = true;
      Main::test = 10;
      java.lang.System::out;Ljava.io.PrintStream()
        .println;Ljava.lang.String;V("init")
    }
  }
  // Default constructor omitted
}
```

Static methods require no further processing, except to call the static ini-

tializer at the top of the method.

Note: Reads of static final fields should not trigger a static initialization (JLS14 §12.4.1), but they do in SJS-JFE since it does not treat static final fields as special at this time. Also, it is impossible at this time to trigger static initialization of a superclass outside of a constructor invocation as a class's static initializer cannot be reliably accessed from outside the class (see the note in section 2.1.3).

Note: Implementing statics in SJS-JFE revealed a bug[10] in the Scala.js version 1.0.1 compiler where static fields were eliminated by the optimizer. The workaround is to disable the optimizer. This bug is fixed in future versions of Scala.js.

2.1.5 Access Control and Namespacing

The Scala.js IR has two "access modifiers": `private` and `public`. These modifiers don't necessarily enforce access control, rather, they namespace member definitions into private and public namespaces. For example, a method call on a private method is invalid unless the call indicates the `private` namespace.

Since JDT checks the validity of the Java program to compile, SJS-JFE does not need to enforce access control. Nevertheless, private methods are dispatched differently than public ones, and so private methods must be put in the `private` namespace. Protected, package-protected, and public methods are emitted in the `public` namespace.

2.1.6 Methods

Methods are transformed into `MethodDef` nodes in the Scala.js IR. Public, protected, and package-protected instance methods are invoked using the `Apply` IR instruction. Private methods are invoked using `ApplyStatically`. Static methods are invoked using the `ApplyStatic` instruction instead of `Apply`.

The Scala.js IR does not provide a classical `return` statement as Java does. Rather, it has a more flexible node, a labeled `return`, that takes the name of a labeled block and steps out of that block, optionally returning a value if the labeled block is used as an expression. It is possible to use a labelled `return` to simulate returning from a method by wrapping the method body in a labeled block. Indeed, SJS-JFE wraps all but the simplest methods in a labeled block:

Java method

```
int meaningOfLife() {
    // ...some code
    return 42;
}
```

Scala.js IR equivalent

```
def meaningOfLife;I(): int = {
    returnScope[int]: {
      // ...some code
      return@returnScope 42
    }
}
```

`returnScope` is a labelled block with a unique name for every method.

If a Java method consists of just a single `return`, SJS-JFE omits the labelled block and simply uses the return expression:

[10] https://github.com/scala-js/scala-js/issues/4021

Java method

```
int meaningOfLife() {
    return 42;
}
```

Scala.js IR equivalent

```
def meaningOfLife(): int = {
  42
}
```

2.1.7 Generics

The Scala.js IR has no concept of generics and uses erased types wherever parameterized types appear. Whenever a value with a generic type is retrieved, it must be cast first to the expected type. This occurs at method invocations, variable references, or field references.

JDT provides type information for erasures, as well as the true bound type of any variable, field, or method whenever it is used. If the returned erased type does not equal the bound type, then the value is wrapped in a cast expression.

2.2 Statements

Statements in JDT extend the `jdt.Statement` abstract class. Statements appear in block bodies, and can wrap expressions.

2.2.1 Assertions

JDT class: `org.eclipse.jdt.core.dom.AssertStatement`

Assert statements are Boolean assertions made using the `assert` keyword.

The Scala.js IR does not support assertions itself, but assertions can be supported by checking the assertion condition and raising an `AssertionError` if it is false.

The semantics of assertions requires some consideration on how to implement them. Normally, assertions are enabled on the JVM by passing the `-ea` command-line flag, but compiling Java to the Scala.js IR raises more questions. Should assertions be omitted by default from the emitted IR? Should they be guarded by a flag that can be set at runtime? Should third-party libraries ship with assertions enabled or disabled? It is not clear which solution maximizes compatibility between existing Scala and Java libraries, and for this reason, SJS-JFE does not emit assert statements.

The transformation for assertions would look like:

```
[[ assert condition ]] :=
  if ((![[ condition ]]))
    [[ throw new java.lang.AssertionError() ]]
```

2.2.2 Blocks

JDT class: `org.eclipse.jdt.core.dom.Block`

A block in JDT is a container for zero or more statements. JDT blocks map directly to the `Block` node in the Scala.js AST.

Scala.js blocks are expressions and have a type associated with them. The last statement or expression in a block is its value. This will play a role in some transformations, such as assignments, that carry out a side-effect *and* generate a value.

2.2.3 Break and Continue

JDT classes: `org.eclipse.jdt.core.dom.BreakStatement`
`org.eclipse.jdt.core.dom.ContinueStatement`

Scala.js does not provide neither `break` nor `continue` nodes in the IR, but the semantics of these statements are very similar to those of the labeled `return` statement. A labeled `return` serves to immediately break out of an enclosing labeled block, optionally returning a value. Wherever a `break` or `continue` can be used, it is possible to create a labeled block and use a labeled `return` to exit out of it. Here is a silly example with a `while` loop and how a labeled `return` can be used to emulate a `break`:

With `break`:

```
while (true) {
  if (Math.random() < 0.1)
    break
}
```

With labeled `return`:

```
breakScope: {
  while (true) {
    if (Math.random() < 0.1)
      return@breakScope
  }
}
```

When not given a label, `break` and `continue` exit the nearest break-able or continue-able scope, respectively.

When given a label, `break` and `continue` exit the same-labeled block. Currently, named breaks work everywhere, but named continues are implemented only for `for` loops.

2.2.4 (Super)Constructor Invocation

JDT classes: `org.eclipse.jdt.core.dom.ConstructorInvocation`
`org.eclipse.jdt.core.dom.SuperConstructorInvocation`

Constructor invocations refer to invocations of a constructor from another constructor of the same class. Super constructor invocations refer to invocations of a super constructor from the extending class's constructor. Section 2.1.3 summarizes how SJS-JFE transforms constructors.

2.2.5 Do-While

JDT class: `org.eclipse.jdt.core.dom.DoStatement`

`do-while` loops map more or less directly to the `DoWhile` node in the Scala.js IR. Two labeled statements must be inserted to support `break` and `continue`. The transformation is:

```
[[ do { body } while (condition)) ]] :=
  breakScope: {
    do {
      continueScope: {
        [[ body ]]
      }
    } while ([[ condition ]])
  }
```

`breakScope` and `continueScope` are synthetic labeled blocks with unique label names. See section 2.2.3.

2.2.6 Empty Statements

JDT class: `org.eclipse.jdt.core.dom.EmptyStatement`

Empty statements (no-ops) map directly to a `Skip` node in the Scala.js IR. `Skip` nodes simply emit a `/* comment */` in the Javascript that is removed by the Scala.js optimizer.

2.2.7 Enhanced `for` Loops

JDT class: `org.eclipse.jdt.core.dom.EnhancedForStatement`

Enhanced `for` loops are ones that contain an `Iterable` or array, like `for (x : iterableOrArray) {}`.

SJS-JFE does not yet implement enhanced `for` loops.

A hypothetical transformation for an array iteration:

```
[[ for (Type x : array) { body } ]] :=
  [[ for (int $i = 0, Type[] $a = array; $i < $a.length; $i++) {
    // $i, $a must be fresh names
    Type x = $a[$i];
    body;
  } ]]
```

A hypothetical transformation for an `Iterable` iteration:

```
[[ for (Type x : iterable) { body } ]] :=
  [[ for (Iterator<Type> $i = iterable.iterator(); $i.hasNext();) {
    // $i must be a fresh name
    Type x = $i.next
    body;
  } ]]
```

These transformations are similar to how enhanced `for` are compiled to JVM bytecode.

See section 2.2.9 for the transformation rule for `for` loops.

2.2.8 Expression Statements

JDT class: `org.eclipse.jdt.core.dom.ExpressionStatement`

Expression statements wrap an instance of `jdt.Expression` that occurs in a statement position. See Section 2.3 for treatment of expressions.

2.2.9 `for` Loops

JDT class: `org.eclipse.jdt.core.dom.ForStatement`

Scala.js provides no way to express traditional `for` loops as in Java. Therefore, they must be transformed to a `while` loop. A `for` loop has four components: a list of initializers, a condition, a list of updaters, and a body. SJS-JFE transforms a `for` loop like so:

```
[[ for(initializers; condition; updaters) { body } ]] := {
  [[ initializers ]]
  breakScope: {
    while ([[ condition ]]) {
```

```
      continueScope: {
        [[ body ]]
      }
      [[ updaters ]]
    }
  }
}
```

`breakScope` and `continueScope` are synthetic labeled blocks with unique label names. See section 2.2.3.

If the `for` statement is nested immediately inside a labeled statement, the labels for `breakScope` and `continueScope` are marked with the label's name. Then, `break` and `continue` statements appearing inside can refer to the labelled block by name. For example:

Java code:

```
FOO: for (; true;) {
    break FOO;
}
```

Scala.js IR:

```
_namedbreak_FOO: {
  while (true) {
    _namedcontinue_FOO: {
      return@_namedbreak_FOO (void 0)
    }
  }
}
```

2.2.10 `if` Statements

JDT class: `org.eclipse.jdt.core.dom.IfStatement`

`if` statements map directly to the `If` node in the Scala.js IR.

Like Scala.js, JDT expresses chains of `else if` by nesting them within the previous `else`. Unlike Scala.js, JDT does not ensure that the `else` block is present. If it is absent, a Scala.js `Skip` node (a no-op) is inserted into the body of the `else`.

The `If` node in the Scala.js IR may return a value like an expression, making it behave more like a ternary conditional. In the case of an `if` statement, this value is ignored.

2.2.11 Labeled Statements

JDT class: `org.eclipse.jdt.core.dom.LabeledStatement`

Labeled statements map directly to the `Labeled` node in the Scala.js IR. In Scala.js, these nodes may produce a value, like expressions, but in this case the value is ignored.

If a labeled statement's body is a loop, then the break and continue scopes of the loop are marked with the label's name. Currently, this is only implemented for `for` loops. See section 2.2.9 for an example.

2.2.12 `return` Statements

JDT class: `org.eclipse.jdt.core.dom.ReturnStatement`

`return` statements in Java exit out of the nearest method or lambda expression with an optional value. Just like `break` (see section 2.2.3), it is possible to

emulate this behaviour using labeled `return` statements in the Scala.js IR. A prerequisite is that every method, function, or lambda must wrap its body with a labeled block of the same name as the return, like so:

```
def someMethod() {
  returnScope: {
    return@returnScope someValue
  }
}
```

The label of `returnScope` is unique for every method, function, or lambda.

If a method contains only a single `return` statement in its body, the labeled block is omitted and the method's body becomes the `return` expression (see section 2.1.6).

If no return value is specified (i.e. returning a `void`), `undefined` is returned instead.

2.2.13 `switch` Statements

JDT class: `org.eclipse.jdt.core.dom.SwitchStatement`

SJS-JFE supports only label-style switch cases like `case 0:`, not rule-style cases like `case 0 ->` introduced in Java 12. The reason for this is that the current version of JDT hides rule-style cases behind a preview flag, which is disabled by default and changes some semantics of the compiler when enabled.

`switch` blocks are somewhat difficult to parse. JDT includes a `SwitchStatement` class that encapsulates the body of a `switch` block, but its body is not hierarhical — `case`s and their statements exist in a flat list, so the first order of business is to parse a `switch` block into a list of cases, where each case contains a list of statements belonging to it.

Once this data structure is created, each case maps roughly to an `if`. The expression of the `switch` statement is evaluated, stored in a temporary variable, then zero or more `if-else` blocks are emitted that check equality of the temporary variable to the case expression. A temporary variable is used to ensure that side-effects are performed at most once. The entire construct is wrapped in a labeled block so that a `break` statement may exit the entire `switch` at any point.

Fallthrough is implemented by storing a second temporary variable, initially set to `false`. Whenever a case is entered, the fallthrough variable is set to `true`, unless the case certainly terminates. A case is entered if the `switch` expression equals the `case` expression, or if the fallthrough flag is set. Therefore, once a case is entered, all following cases will be entered unless control exits the `switch` in some way.

A possible optimization that can be made is transforming the `switch` statement into a `Match` in the emitted IR if there is no fall-through, though this is not implemented yet.

Here is an example of a `switch` statement with integers compiled by SJS-JFE:

Java:

```
switch (i % 2) {
  case 0:
    log("even");
    break;
  case 1:
    log("odd");
  default:
    log("fallthrough");
}
```

Scala.js IR:

```
_break_1: {
  val $e: int = (i %[int] 2);
  var $fall: boolean = false;
  if (($fall || $e === 0)) {
    log("even");
    return@_break_1 (void 0)
  };
  if (($fall || $e === 1)) {
    $fall = true;
    log("odd")
  };
  if (($fall || true)) {
    $fall = true;
    log("fallthrough")
  }
}
```

(Some syntactic details were omitted to fit the listing in its column)

2.2.14 Synchronized Blocks

JDT class: `org.eclipse.jdt.core.dom.SynchronizedStatement`

Javascript is single-threaded, therefore `synchronized` statements have little effect. However, a `synchronized` block must take an expression for the lock to acquire. Evaluating the expression may also trigger side-effects. If the expression evaluates to `null`, a `NullPointerException` must be thrown (JLS14 §14.19). This condition must be checked at runtime.

The transformation rule for `synchronized` statements is:

```
[[ synchronized (expression) { body } ]] := {
  val sideEffect: <typeof expression> = [[ expression ]]
  // sideEffect is a fresh name
  if ((sideEffect === null)) {
    throw new java.lang.NullPointerException().<init>;V()
  } else {
    [[ body ]]
  }
}
```

2.2.15 Throw Statements

JDT class: `org.eclipse.jdt.core.dom.ThrowStatement`

`throw` statements more or less map directly to the `Throw` node in the Scala.js IR, though their utility is limited since SJS-JFE does not yet implement `try/catch` statements. If the expression of a `throw` statement evaluates to `null`¸ a `NullPointerException` must be thrown. This condition must be checked at runtime.

The transformation rule for `throw` statements is:

```
[[ throw expression ]] := {
```

```
  val sideEffect: <typeof expression> = [[ expression ]]
  // sideEffect is a fresh name
  if ((sideEffect === null)) {
    throw new java.lang.NullPointerException().<init>;V()
  } else {
    throw sideEffect
  }
}
```

2.2.16 Try/Catch/Finally

JDT class: `org.eclipse.jdt.core.dom.TryStatement`

SJS-JFE does not yet implement `try-catch-finally` statements. Scala.js does provide `TryCatch` and `TryFinally` IR nodes.

Here is a hypothetical transformation for `try-catch-finally`:

```
[[ try { tryBody }
    catch (ExceptionType e) { catchBody }
    // possibly more catches
    finally { finalizer } ]] :=
  try {
    [[ tryBody ]]
  } catch {
    if ((e.isInstanceOf[ExceptionType])) {
      [[ catchBody ]]
    } else {
      // Handle other catches
    }
  } finally {
    [[ finalizer ]]
  }
```

The `try-catch-finally` construct is expressed in IR nodes like so:

```
TryFinally(
  TryCatch(
    [[ tryBody ]],
    [[ catches ]]
  ),
  [[ finalizer ]]
)
```

2.2.17 Type Declarations

JDT class: `org.eclipse.jdt.core.dom.TypeDeclarationStatement`

Type declaration statements represent local (not class-level) definitions of classes or enums. SJS-JFE does not yet implement local type declarations, due to non-trivial handling of captures.

2.2.18 Variable Declarations

JDT class: `org.eclipse.jdt.core.dom.VariableDeclarationStatement`

Variable declarations represent declarations and initializations of local variables (not fields) in a statement position. Variable declaration *expressions* also exist; see section 2.3.25.

Java allows several variables of the same type to be declared and initialized in the same statement, for instance: `int i = 0, j = 1`. Each "fragment" contains a name and optionally an initializer, and one `VarDef` IR node is emitted per fragment.

2.2.19 `while` Loops

JDT class: `org.eclipse.jdt.core.dom.WhileStatement`

`while` loops map more or less directly to the `While` node in the Scala.js IR. Two labeled statements must be inserted to support `break` and `continue`. The transformation is:

```
[[ while (condition) { body } ]] :=
  breakScope: {
    while([[ condition ]]) {
      continueScope: {
        [[ body ]]
      }
    }
  }
```

`breakScope` and `continueScope` are synthetic labeled blocks with unique label names. See section 2.2.3.

2.2.20 `yield` Statements

JDT class: `org.eclipse.jdt.core.dom.YieldStatement`

`yield` statements are not yet implemented in SJS-JFE.

2.3 Expressions

Expressions may contain other expressions, and generally represent a value. If an expression is a block-like element such as a lambda or anonymous class, it may contain other statements as well.

2.3.1 Annotations

JDT class: `org.eclipse.jdt.core.dom.Annotation`

SJS-JFE does not support annotations. When it encounters an annotation expression, it emits a no-op.

2.3.2 Array Accesss

JDT class: `org.eclipse.jdt.core.dom.ArrayAccess`

Array access expressions correspond exactly to the `ArraySelect` nodes in Scala.js. They both take an array identifier and an integer index as properties.

2.3.3 Array Creation

JDT class: `org.eclipse.jdt.core.dom.ArrayCreation`

Array creation expressions represent arrays created with the `new` keyword. There are two ways this can happen. If an array initializer is given, the array is created via the `ArrayValue` node in the Scala.js IR (see below). If no array initializer is given, the array is created with `NewArray`, which creates a zeroed array of the appropriate dimension(s).

2.3.4 Array Initializer

JDT class: `org.eclipse.jdt.core.dom.ArrayInitializer`

Array initializer expressions create arrays by specifying their elements delimited with { curly braces }. They map directly to the `ArrayValue` node in the Scala.js IR.

2.3.5 Assignments

JDT class: `org.eclipse.jdt.core.dom.Assignment`

Assignments take the form `lhs operator rhs`. `operator` may be any of the following symbols: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, or `>>>=`. In the case of an arithmetic assignment, SJS-JFE treats it as a simple assignment by constructing the appropriate arithmetic operation node for the right-hand side; e.g. `a += 1` is interpreted as `a = a + 1` (see section 2.3.12).

If the left-hand-side of the assignment is a static field, the field's synthetic setter is invoked with the right-hand-side as the argument (see section 2.1.4). Otherwise, SJS-JFE emits the `Assign` node, which takes the left-hand and right-hand sides of the assignment as arguments.

There is one additional consideration: in Java, assignments can be used in the expression position, and evaluate to the assigned value. This makes it possible to "chain" assignments like `int a = b = c = 10`. Scala.js does not provide this feature, so whenever an assignment is used in the expression position, an extra read is performed on the left-hand-side and its value returned. For example, the Java statement `a = b = 10;` would roughly be translated to the following IR: `Assign(a, Block(Assign(b, 10), VarRef(a)))`. The general transformation for an assignment in an expression position is:

```
[[ name = value ]] := {
  name = value;
  name
}
```

for instance fields and local variables. For static fields, it is:

```
[[ name = value ]] := {
  Class::name_$eq;X;V(value) // Invoke the setter
  Class::name;X()            // Invoke the getter
}
```

where `X` represents the type of the static field and `Class` is the field's declaring class name.

2.3.6 Casts

JDT class: `org.eclipse.jdt.core.dom.CastExpression`

Cast expressions take the form `(Type) expr`. They map directly to the `AsInstanceOf` node in the Scala.js IR.

2.3.7 Class Instance Creation

JDT class: `org.eclipse.jdt.core.dom.ClassInstanceCreation`

Class instance creation nodes represent instantiation of classes using the `new` keyword. JDT automatically provides a binding to the correct constructor, including type information for parameters.

Most JDK classes, and all user-defined classes, are instantiated with the `New` node in the Scala.js IR. This node behaves like a simplified method invocation, taking as arguments the name of the class, the signature of the constructor, and a list of arguments to the constructor.

Boxed types (`java.lang.{Boolean, Character, Byte, Short, Integer, Long, Float, Double}`), as well as `java.lang.String`, are called *hijacked classes* in Scala.js, and have special implementations. Instances of these classes must be created by calling the synthetic static method `::new()` with the same arguments as the constructor. SJS-JFE automatically emits the correct Scala.js IR node (`ApplyStatic` or `New`) based on whether the class is hijacked or not. Since all hijacked classes are marked as `final` (thus the set of classes that must be instantiated with a static call is finite and fixed), this is a static transformation that can be safely made at compile-time.

If the instantiated class is an inner class, a handle to its outer scope (`this`) is provided as a synthetic argument to the constructor.

2.3.8 Conditional (Ternary) Expressions

JDT class: `org.eclipse.jdt.core.dom.ConditionalExpression`

Ternary expressions take the form `cond ? thenExpr : elseExpr`. They map directly to the `If` node in the Scala.js IR. Recall, the `If` node takes *then* and *else* expressions and evaluates the appropriate one based on the value of the condition, making it semantically equivalent to the ternary operator rather than Java's `if-else`.

2.3.9 Creation References

JDT class: `org.eclipse.jdt.core.dom.CreationReference`

Creation references are method references to a class's constructor, like `Thing::new`. They are not implemented yet in SJS-JFE.

2.3.10 Expression Method References

JDT class: `org.eclipse.jdt.core.dom.ExpressionMethodReference`

Expression method references are references to methods with an explicit qualifier, like `qualifier::method`. They are not implemented yet in SJS-JFE.

2.3.11 Field Access

JDT classes: `org.eclipse.jdt.core.dom.FieldAccess`
`org.eclipse.jdt.core.dom.SuperFieldAccess`

JDT provides several AST nodes to represent field accesses. The `FieldAccess` node is the most flexible.

`FieldAccess` nodes describe a qualifier and a field name. The qualifier need not be an identifier — in fact, in cases where the qualifier is *not* and identifier (e.g. the `this` keyword or a method call), such field access can only be expressed with a `FieldAccess` node. When the qualifier is an identifier, the `FieldAccess` and `QualifiedName` AST nodes can be used interchangeably[11].

At the moment, SJS-JFE maps `FieldAccess` directly to the `Select` node in the Scala.js IR. This does not entirely satisfy the semantics for `FieldAccess`. In practice, JDT seems to prefer the `QualifiedName` AST node for field accesses. Further testing is required to identify pathologic cases, and the appearance of these cases may vary depending on the JDT version used.

For example, the expression `instance.field` is a field access on an object `instance` for field `field`. If `instance` is defined in an enclosing lexical scope, this would not map directly to a `Select` IR node — the qualifier `instance` would have to be prepended with one or more field accesses to get the appropriate enclosing scope. Field accesses using `QualifiedName` have the correct behaviour. However, we are unable to find an example of a Java source where JDT prefers `FieldAccess` over `QualifiedName`, so the implementation of `FieldAccess` is conservatively left incorrect for now, pending further investigation.

Field accesses on superclasses can only be expressed with the `SuperFieldAccess` node. This, too, maps directly to the `Select` node in the Scala.js AST, except that the `Select` is given the class name of the superclass.

2.3.12 Infix Expressions

JDT class: `org.eclipse.jdt.core.dom.InfixExpression`

Infix expressions are binary expressions using the arithmetic operators `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `<<`, `>>`, `>>>`, and logical operators `==`, `!=`, `>`, `>=`, `<`, `<=`, `&&`, and `||`.

JDT allows for more than 2 operands to be specified in an infix expression (as long as they have the same operator) to limit the depth of the AST. SJS-JFE treats infix operators as binary functions, so when more than 2 operands are given, they are folded and the transformation is applied to two operands at a time.

If the operator is `+` and the left operand is a string type, SJS-JFE performs a string concatenation using the `BinaryOp` node in the Scala.js IR. Otherwise, it constructs a binary operation using a similar process as the Scala.js compiler. This algorithm is summarized here:

1. First, the result type of the operation is inferred. If either operand is a boxed type, the operand is unboxed.
    - If either the left or right operand is a `double`, the result type is `double`. If not,

[11] The JDT javadoc on the `FieldAccess` class provides a decent explanation of these cases

- If either operand is a `float`, the result type is `float`. If not,
- If either operand is a `long`, the result type is `long`. If not,
- If either operand is an `int`, `char`, `byte`, or `short`, the result type is `int`. If not,
- If either operand is a `boolean`, the result type is `boolean`.
- If none of the above cases apply, the result type is `Any`.

2. The left operand is cast to the output type.
3. The right operand is cast to the output type, unless the operation is a bitshift, in which case the right operand is cast to `int`.
4. Depending on the output type, the correct Scala.js binary operator is chosen (the `+` operator for integers is different than the one for floats, for example).
5. A `BinaryOp` IR node is emitted with the chosen operator and casted operands.

2.3.13 Instanceof Expressions

JDT class: `org.eclipse.jdt.core.dom.InstanceofExpression`

`instanceof` expressions are mapped directly to `IsInstanceOf` nodes in the Scala.js IR.

2.3.14 Lambda Expressions

JDT class: `jdt.LambdaExpression`

Lambda expressions aren't implemented in SJS-JFE yet due to the non-trivial handling of captures.

2.3.15 Literals

JDT classes: `org.eclipse.jdt.core.dom.BooleanLiteral`
`org.eclipse.jdt.core.dom.CharacterLiteral`
`org.eclipse.jdt.core.dom.NullLiteral`
`org.eclipse.jdt.core.dom.NumberLiteral`
`org.eclipse.jdt.core.dom.StringLiteral`

Transforming literals depends on the type of literal.

- For Boolean, character, and string literals, JDT provides the typed literal value. These are transformed into `BooleanLiteral`, `CharLiteral`, and `StringLiteral` nodes in the Scala.js IR, respectively.
- Null literals are transformed to the `Null` node in the Scala.js IR.
- Number literals encompass double, float, integer, and long literals. Number literal classes provide only the string token of the literal, which is not very useful since it must be parsed. However, JDT can resolve the constant expression value of the literal, which is cast to the appropriate type and transformed to `DoubleLiteral`, `FloatLiteral`, `IntLiteral`, or `LongLiteral` nodes in the Scala.js IR.

2.3.16 Method Invocations

JDT classes: `org.eclipse.jdt.core.dom.MethodInvocation`
`org.eclipse.jdt.core.dom.SuperMethodInvocation`

JDT resolves method bindings for all method invocations. The binding includes information about the method's return type (including erasures), parameter types, receiver, etc. The proper transformation can be made on the basis of binding information:

- If the method is static, it is invoked with `ApplyStatic`.
- If the method is not static, has no receiver, and is declared in an enclosing lexical scope, it is invoked with `Apply` using a reference to the appropriate enclosing instance as the receiver.
- If the method is not static and has no receiver, it is invoked with `Apply` (if the method is public) or `ApplyStatically` (if private) using `this` as the receiver.
- If the method is not static and declares a receiver, it is invoked with `Apply` (if public) or `ApplyStatically` (if private) using the receiver.
- If the method is not static and the receiver is `super`, it is invoked with `ApplyStatically`, since the method may have been overridden. This case is represented by `SuperMethodInvocation` in the JDT AST.

The method may be parameterized, in which case the return type of the method is erased, and must be casted to the expected type. Fortunately, JDT provides information about the erased type and the invocation type in the method binding. If the erased type does not match the return type of the particular invocation, the invocation is wrapped in a cast.

2.3.17 Parenthesized Expressions

JDT class: `org.eclipse.jdt.core.dom.ParenthesizedExpression`

Parenthesized expressions are simply expressions occurring within `(` parenthesis delimiters `)`. They receive no special treatment and the expression within the parentheses is transformed normally.

This trivial transformation is `[[ (expr) ]] := [[ expr ]]`

2.3.18 Postfix Expressions

JDT class: `org.eclipse.jdt.core.dom.PostfixExpression`

Postfix expressions encompass only increment and decrement operations. They can appear in both statement and expression positions, and SJS-JFE emits different IR trees for either possibility.

If the expression is used as a statement, SJS-JFE emits a simple assignment with the appropriate addition/subtraction.

If the expression is used as an expression, SJS-JFE stores the original value in a temporary variable, does the assignment, then returns the temporary value. The full transformation (for increment; decrement is similar):

```
[[ a++ /* in expression position */ ]] := {
  val temp: T = a;
  a = (a +[T] 1);
  temp
}
```

where `temp` is a fresh name and `T` is the primitive type of `a`. Boxed operands are unboxed if needed.

2.3.19 Prefix Expressions

Prefix expressions encompass increment, decrement, complement (`~`), Boolean negation (`!`), unary minus, and unary plus operations.

Increment and decrement. Increments and decrements can appear in both statement and expression positions, and SJS-JFE emits different IR trees for either possibility. If the expression is used as a statement, SJS-JFE emits a simple assignment with the appropriate addition/subtraction. If the expression is used in an expression position, SJS-JFE does the assignment, then reads the left-hand side of the assignment and returns it value.

Complement To quote JLS14 §15.5.5: "In all cases, `~x` equals `(-x)-1`. However, SJS-JFE uses the equivalent `x XOR -1` formula, which simply flips all bits in the operand.

Boolean negation. Emits a `UnaryOp` node in the Scala.js IR with the Boolean `!` operator.

Unary minus Emits a `BinaryOp` node in the Scala.js IR that subtracts the value from the integer value zero.

Unary plus Emits a `BinaryOp` node in the Scala.js IR that adds the value to the integer value zero. This seemingly useless arithmetic nevertheless unboxes the argument and promotes it to integer width.

2.3.20 Qualified and Simple Names

JDT classes: `org.eclipse.jdt.core.dom.QualifiedName`
`org.eclipse.jdt.core.dom.SimpleName`

To quote the JDT doc: "A simple name is an identifier other than a keyword, Boolean literal, or null literal". A qualified name is a simple name qualified by either a simple or qualified name. Qualifiers must be names; otherwise, the `FieldAccess` AST node must be used.

Qualified names. Let `v` be the named variable.

If the name is `length` and the qualifier is an array type, SJS-JFE emits an `ArrayLength` IR node.

If `v` is static, then the qualifier must be a class name. The class must be top-level or a static inner class (since non-static inner classes cannot have static

members), which has a globally-resolvable name. SJS-JFE emits a method invocation of `v`'s getter.

If `v` is not static, then the qualifier must be an instance of an object and `v` must be a field. SJS-JFE emits a `Select` IR node to access the field.

Simple names. Simple names are somewhat more complicated because they can refer to all local variables, fields, and methods in scope, including scopes of lexically enclosing instances. Again, let `v` be the named variable.

If `v` is a field and is static, SJS-JFE emits a static getter invocation using the field's declaring class as a qualifier.

If `v` is an instance field, SJS-JFE emits a `Select`, whose qualifier is the appropriate selector of the instance where `v` is defined (or `this` if it is a field on `this`).

If `v` is a local variable or method parameter, SJS-JFE emits a `VarRef` IR node.

The field or variable referred to by the name may be parameterized. If the erasure of the variable doesn't match the actual type, SJS-JFE wraps the entire access in a cast to convert to the expected type.

2.3.21 Switch Expressions

JDT class: `org.eclipse.jdt.core.dom.SwitchExpression`

Switch statements were introduced as a preview feature in JLS 12 and as a full feature in JLS 14 (JLS14 §15.28). SJS-JFE doesn't support switch statements in the expression position yet (see section 2.2.13).

2.3.22 `this` Expressions

JDT class: `org.eclipse.jdt.core.dom.ThisExpression`

`this` expressions represent the `this` keyword as well as the qualified `this` referring to an enclosing lexical instance.

If the expression is unqualified, the expression maps directly to the `This` node in the Scala.js IR. If the expression is qualified, then a chain of field accesses is generated to retrieve the appropriate instance. Recall, inner classes have a synthetic field that refers to the parent lexical instance. Any ancestor scope can be accessed by traversing these fields.

2.3.23 Type Literals

JDT class: `org.eclipse.jdt.core.dom.TypeLiteral`

Type literal expressions take the form `Type.class`. `Type` may be a class, primitive, or `void`. Type literals map directly to the `ClassOf` node in the Scala.js IR.

2.3.24 Type Method References

JDT class: `org.eclipse.jdt.core.dom.TypeMethodReference`

Type method references are references to methods on types, like `String::valueOf`. They are not implemented yet in SJS-JFE.

2.3.25 Variable Declaration Expressions

JDT class: `org.eclipse.jdt.core.dom.VariableDeclarationExpression`

Variable declaration expressions are equivalent to variable declaration statements (section 2.2.18), except appearing in an expression position. Most commonly, variable declaration expressions are used as initializers in `for` loops.

2.4 Testing transformations

SJS-JFE is primarily tested using functional tests that focus on a single language feature at one time. In a functional test, a short but complete Java program is written that is compiled by SJS-JFE, then linked using the Scala.js compiler with the Scala.js libraries before being executed in a NodeJS environment. The tested program prints values to the standard output. A test passes if the output equals exactly an expected sequence of values.

Individual language features are tested in isolation. A step forward is to write a suite of integration tests that combine several language features together into a useful program. A classic example would be to emulate a Turing machine and trace its execution. Another important milestone in making SJS-JFE ready for adoption is to select a few well-known Java libraries, compile them to Scala.js IR, and publish artifacts containing the IR files to a public repository.

SJS-JFE's tests contain nearly as many lines of code as the main transpiler. Great care was taken to test as many code paths as possible, which is why many language features were not implemented in time — it is time-consuming to not only formulate and write the appropriate transformations, but write tests and explore different edge cases. Still, many edge cases are probably missing from tests; Particularly challenging tests were those regarding handling of different primitive types and implicit value conversions. Testing every language feature in every possible context is a Herculean task, so a potential improvement would be to randomly (if not exhaustively) generate many test cases with different parameters, or at the very least, fuzz the values used in the tests so as to catch the occasional well-hidden bug.

3 Conclusion and Future Directions

SJS-JFE supports enough Java language features to enable writing complex programs, but there is still no shortage of work to do if SJS-JFE should be widely adopted. First and foremost, the remaining Java features must be implemented. Of some concern are local classes, anonymous classes, and lambdas, which must handle captures. Unfortunately, JDT does not provide a list of captures for these constructs. Capture handling must also integrate with other features, like enclosing scope resolution, meaning that this feature will affect SJS-JFE's code in several different places.

It is known that there are 2 unsolved problems in computer science: naming things, cache invalidation, and off-by-one errors. True to this joke, arguably one of the most difficult tasks is to come up with a more catchy, yet descriptive name for SJS-JFE, and maybe an attractive logo.

As mentioned in section 2.4, SJS-JFE deserves to be tested more thoroughly, to demonstrate that all language features are handled correctly in as many different contexts as possible.

A possible optimization that can be made in the future is resolution of constant expressions. JDT provides quite sophisticated resolution for values, including those that are guaranteed never to change through a program's execution. These constant expressions can be inlined into the IR for an improvement in speed and IR size. It is unclear whether JDT's constant expression resolution provides better advantages than Scala.js's own optimizer, but at the very least, constant expression inlining would make compiling Java ASTs to IR somewhat faster.

Scala.js programs written in Scala have access to JS APIs, most notably the JS standard library and the DOM in browsers. These APIs allow programs to interact with the world around them. Java programs compiled with SJS-JFE do not have access to these APIs, so to write a useful browser application using only Java would be impossible.

The greatest hurdle to overcome, however, is developing convenient tooling around SJS-JFE. At the moment, SJS-JFE is best invoked through the test harness — it does not have a command-line interface. Also, SJS-JFE require several environment variables to be set, notably the paths to the Scala.js libraries, for programs to link correctly. These variables are set programmatically and can't be modified. SJS-JFE should also include the proper tooling, such as an `sbt` plugin, that automatically detects and compiles Java sources into Scala.js IR, so that using Java in a Scala.js project is as ease as using Java code in a Scala project on the JVM.

References

[1] Sébastien Doeraene. “Cross-Platform Language Design”. PhD thesis. Lausanne, Switzerland: École Polytechnique Fédérale de Lausanne, June 2018.

[2] James Gosling et al. *The Java ® Language Specification, Java SE 14 Edition*. Tech. rep. Oracle Corporation, Feb. 2020.