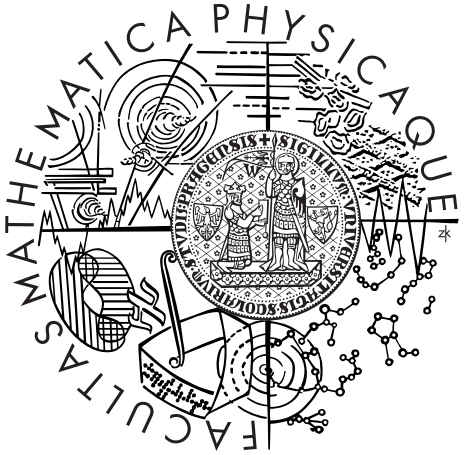
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Jan Široký

Scala Web Application Toolkit

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Pavel Ježek

Study program: Computer Science

Specialization: Software Systems

Prague 2013

I would like to thank my family for continuous support during my work on the thesis, my supervisor Pavel Ježek for invaluable consultations and advices he provided me with and my friend Jirka Helmich for proofrading of the thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Scala Web Application Toolkit

Autor: Jan Široký

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Pavel Ježek, Ph.D, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Práce nejdříve popisuje nevýhody JavaScriptu v kontextu rozsáhlých webových aplikací a představuje, jak se tyto problémy obvykle řeší pomocí kompilátoru z jiného vyššího programovacího jazyka do JavaScriptu. Práce dále ukazuje, jaké problémy je nutné řešit v rámci implementace takového kompilátoru a jak překonat další překážky (interoperabilita, distribuce knihoven) vyplývající z toho, že programy nejsou napsány přímo v JavaScriptu, nýbrž ve Scale. Na vyvinutém kompilátoru a jednoduchém běhovém prostředí pak dále staví infrastrukturu, která umožňuje vzdálené volání metod (RPC). Tato infrastruktura zahrnuje mimo jiné i dynamické načítaní skriptů ze serveru a (de)serializér grafů objektů. V závěru pak porovnává vyvinutý toolkit s jinými podobnými projekty a představuje několik zajímavých směrů, kterými by se celý projekt dále mohl ubírat.

Klíčová slova: Scala, JavaScript, RIA, JSON, RPC

Title: Scala Web Application Toolkit

Author: Jan Široký

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D, Department of Distributed and Dependable Systems

Abstract: The thesis describes disadvantages of JavaScript in the context of big web applications and presents how they can be eliminated by compilation from a different language to JavaScript. With Scala as the source language, the thesis shows what it takes to implement such a compiler and how to solve issues that arise with the compilation process (interoperability, library distribution). On top of the compiler with a simple runtime a remote metod invocation (RPC) infrastructue is built. It also involves a dynamic client side script loader and an object graph (de)serializer. Finally the work compares the implemented toolkit to similar projects and proposes several interesting directions the toolkit can evolve into.

Keywords: Scala, JavaScript, RIA, JSON, RPC

Contents

1. Introduction

The environment of websites and web applications has been rapidly evolving and going through a lot of changes for the last twenty years. It may even be considered the most dynamic area of software development, currently together with mobile application development. The first websites were just static HTML pages linked together with anchors. But people soon found out it is possible to generate the pages on the server, thus provide a dynamic behavior and mimic standard desktop applications. The obvious drawback is that when a user performs an action the whole page has to be reloaded.

In order to mitigate the problems of dynamic pages, new browser based technologies started to emerge. The most known are Flash [1], Java Applets [2] and JavaScript [3] [4]. The proprietary Flash and Java Applets were designed solely for the purpose of web application development, JavaScript was in the first phases meant as a simple scripting language for a manipulation with a web page. However, with the ability to send HTTP requests (known as AJAX [5]), it turned out that JavaScript may be even used as a platform for web applications.

The current tendency is to move from the proprietary technologies to JavaScript, which is widely supported by browsers and devices. With the new HTML5 [6] specifications, which are being adopted quite quickly by the major browsers, even the advantages of other technologies (e.g. video in Flash) are disappearing. So a growing number of web applications is at the moment designed as a single HTML page with embedded JavaScript that takes care of everything. The single page is backed by a server API that the client JavaScript communicates with. This architecture is actually quite similar to the desktop applications therefore design patterns and techniques that are already proven-right from the desktop environment may be utilized. From the architectural point of view the web and desktop application branches are merging.

During the past couple of years, JavaScript has been used so extensively that the developers started to reach its limits in modularization and maintainability, mainly while working on big applications. Other programming languages however handle those issues better and because JavaScript is a quite powerful programming language, it is not that difficult to use it as a compilation target for those languages. Even completely new programming languages with the only purpose to improve JavaScript appeared, so there is currently more than one hundred of languages that compile to JavaScript [7], both new and mainstream like Java, C#, C++ etc.

This approach is actually similar to what happened in the world of standard programming languages where the bottom level is an assembly language. Other more expressive languages like C or Pascal which compile to an assembler emerged as soon as an assembly language was not sufficient enough.

1.1 Motivation

The vast amount of compilers targeting JavaScript suggests there actually are objective reasons and motivation for them, which drive their development efforts. And because compilation is a development step that is not present during development of plain JavaScript programs, the motivation has to be strong enough so programmers are willing to bear this additional step. The following sections should summarize the main disadvantages of JavaScript and briefly describe current trends in the area of JavaScript improvements and compilation into it.

1.1.1 JavaScript Disadvantages

When evaluating a programming language, it is always necessary to specify the context of its usage. Here we are interested in the area of large and complex web applications so it does not mean that JavaScript should not be used for small and even middle-sized applications.

Language

The JavaScript itself is a dynamically typed language with prototype-based inheritance. It has no notion of classes, modules, packages or any other similar concepts. The whole program consists of bunch of functions and nested objects, which are all accessible from the global scope due to the fact that object is a key-value table without any access modifiers. Also there is not any way how to define an interface and its implementation, so it is impossible to hide complexity of components behind interfaces. All this leads to a drawback that JavaScript programs are difficult to modularize. Advantages of static typing over dynamic typing are nicely summarized by Erik Meijer [8]:

> "...earlier detection of programming mistakes (e.g. preventing adding an integer to a boolean), better documentation in the form of type signatures (e.g. incorporating number and types of arguments when resolving names), more opportunities for compiler optimizations (e.g. replacing virtual calls by direct calls when the exact type of the receiver is known statically), increased run-time efficiency (e.g. not all values need to carry a dynamic type), and a better design time developer experience (e.g. knowing the type of the receiver, the IDE can present a drop-down menu of all applicable members)."

Even though inheritance is present in JavaScript, it is very different from the class-based inheritance most of the programmers are used to. So it takes some time to fully understand it and take advantage of it. The JavaScript community realizes some of the disadvantages and tries to remedy them on the language level using sometimes peculiar constructs (for example hiding variables and functions by declaring them as local inside body of an anonymous function which is immediately executed). But there is no standardized or widely adopted way so those efforts stay in the position of suggested conventions.

Distribution

The most common way of library distribution is one big JavaScript file containing everything. Therefore when a fraction of a library is used, the whole library has to be loaded into the page. Dependencies among the libraries are described in the documentation, thus absence of a dependency causes run-time error claiming that a function or an object is not defined. This leads to an environment where the libraries are more likely monolithic and do not share the common functionality even if they could. Similar problem arises when a single application is subdivided into multiple source files. Then all the files have to be declared in the page using script elements in their dependency order. There already exist loaders that handle packaging (most commonly used RequireJS [9], HeadJS [10]) but all share the common disadvantage that the organization of source files into modules and dependencies among them has to be explicitly declared.

Development Process

As a consequence of the dynamic typing and the fact that field values and therefore method implementations are resolved at run-time, some of the processes that can be performed automatically by an IDE or an external tool are not as powerful as in statically typed languages or are even unusable. Because code analyzers and optimizers have much less information about the programs and cannot easily reason about them, their power is weaker. There is however couple of such tools, e.g. code quality analyzers JSLint [11] and JSHint [12] and code optimizer Google Closure Compiler [13]). Automatic refactoring cannot be utilized at all, so in real life it boils down to the unreliable operation *"replace in files"*. Intellisense in IDEs is currently at least partially usable however still far behind statically typed languages. Hand in hand with intellisense comes the source documentation in a form of documentation comments, because when it does not pop up in the IDE, programmers are less motivated to write it.

1.1.2 Improving JavaScript

When examining current directions of JavaScript improvement, most of them have couple of things in common. They use source languages with static (or at least optionally static) type systems in order to solve the issues related to development process. Moreover they introduce the concept of packages (or modules, namespaces) for better modularization of programs. All of them switch the prototype-based inheritance for class-based inheritance and together with classes, majority of them also introduces interfaces (or traits, mix-ins) and access modifiers.

Using such a source language that is compilable to JavaScript fixes the aforementioned issues. The problems related to the distribution of programs and libraries are mostly handled at the run-time, so it does not affect the source language.

Ideally the source language should be much more expressive than JavaScript, not only small superset in terms of language features, so the compilation step

really pays off. Similarly to the step from an assembly language to C, the C language is not just another assembler with some new instructions, but it offers new paradigms, control structures and language constructs that are completely foreign to an assembler.

New Languages

One branch of the improvement process leads to a design of new programming languages solely with the purpose to extend JavaScript. The most well-known examples are Dart [14], CoffeeScript [15] and TypeScript [16]. Their advantage is that they are designed with the knowledge that programs would be compiled to JavaScript so for example primitive types do not deviate from JavaScript primitive types. And some of the languages allow optional typing, which is good for interoperability with existing libraries. Another benefit is that from the syntax point of view they are mostly quite similar to JavaScript so the learning curve is flat for established web application developers.

On the other hand, the fact they are brand new implies that for the whole application, one has to be familiar with three programming languages at a time. One for server side, the new language for client side and JavaScript for debugging. Currently both the compilers and browsers start to support source maps which enable debugging in the source language instead of JavaScript, but the need of JavaScript knowledge will be still present.

Type systems of those languages are primitive compared to Haskell, Scala or even C# and their expressiveness is close to JavaScript. They do not bring any revolutionary concepts therefore the improvement is not as big as it theoretically could be. From that point of view, CoffeeScript is however the most advanced one.

Existing Languages

The more appealing idea is to use existing proven language and add the JavaScript backend to its compiler. And if the language is well-suited for web applications or APIs, then even better because the two ultimate aims could be accomplished: having the whole application written in one language and code-sharing and reuse between server-side and client-side. These goals are so attractive that the project Node.js [17] tackles them from the other side by enabling usage of JavaScript on the servers.

Having both sides of the application written in one language even allows implementation of middleware infrastructures that would support for example remote method invocation known as RPC [18]. That is unthinkable in the plain JavaScript world.

Drawback is that the languages differ from JavaScript a lot, interoperability with it is more inconvenient and in some cases they are less expressive than JavaScript (e.g. lack of closures in Java). The most famous representatives of this branch are

GWT (Google Web Toolkit [19]) for Java and SharpKit [20] for C#, but almost all established languages have their JavaScript compilers.

1.1.3 Why Scala

During the development of one web application in Scala whose big part should run in the web browser, it was decided that it would be nice to implement that part using something like GWT, just for Scala. However no such tool turned out to be both usable, maintained and not abandoned. That lead to an origin of the project to implement such a tool for Scala. Later it emerged that it would be more likely a set of tools, therefore the name of the project and also title of this thesis converged to "Scala Web Application Toolkit", shortly "Swat".

Scala is relatively new yet already established programming language. Among the family of modern programming languages that run on top of Java Virtual Machine (i.e. Scala, Groovy, Clojure, Kotlin) it is the most popular one according to TIOBE index [21] as of writing this thesis, currently gaining a lot of momentum and being already adopted by commercial companies like Twitter or LinkedIn. Moreover, it is backed by strong academic community so the language with all the concepts and paradigms are well thought through. All these facts determined that it would be good idea to start the project because there is a lot of possible potential.

1.2 Project Aim

It may be already apparent from the motivation, but the main aim of the project is to create a Scala to JavaScript compiler. From the topmost level point of view it should take Scala code on the input and produce JavaScript code on the output. The generated code should be somehow embedded or loaded into a web page so when the page is displayed, it runs and on the algorithmic level does the same thing as if the original Scala code was executed on the JVM. Moreover, it should be possible to access both native JavaScript APIs (e.g. DOM [22], browser-related APIs) and other already existing JavaScript libraries (e.g. jQuery [23]) from the Scala code in order to manipulate with the web page or with the browser.

With the described setup, it should be possible to take advantage of the fact that both client and server are written in one language and implement simple RPC mechanism. The obvious implication is that objects would have to be sent through HTTP protocol in both directions and therefore serialized to some common format. To accomplish that, a client side (de)serializer and a server side (de)serializer has to be implemented.

To make the development simple and user-friendly, the tool should be integrated into the SBT (Scala Build Tool [24]) which is currently the most used and to some extent an industry standard in the Scala ecosystem. Moreover, it should be relatively straightforward to integrate the Swat infrastructure into any web application framework. For the purpose of an example, the web application framework

Play [25] will be used as a reference. SBT and Play are supported by the Typesafe [26] which is a company founded by the authors of Scala so both projects are highly maintained with big communities around them.

Finally, a sample application that demonstrates usage of the toolkit should be created. Because the whole toolkit is quite a complex piece of software whose size spans out of the scope of this thesis, it is presupposed that not all aspects of it would be fully completed. However, the most challenging and difficult parts should be finished. The areas that are not so interesting from the architectural or design point of view would be given lower priority. So the sample application will be rather small.

1.2.1 Goals

To get better understanding of what has to be done and to somehow break down the project, it is possible to identify couple of steps that have to be analyzed, decided and possibly implemented. The first set of core goals that are critically necessary for the toolkit consists of the following:

1 - Compiler The Scala to JavaScript compiler and its integration to SBT. It should be the most extensively tested component of the toolkit.

2 - Interoperability A way how to interoperate with JavaScript APIs and libraries.

3 - Runtime When the generated code would not be executable by itself, a runtime that would support the execution.

4 - Libraries Provide subset of the Scala and Java standard libraries relevant to the web environment.

5 - Integration Straightforward means of toolkit integration to a web application framework.

6 - Distribution In the ideal case a distribution method where a distribution unit contains both Scala bytecode and the generated JavaScript code.

When the core goals are finished it is possible to start working on other tasks that the toolkit in principle could do without. They bring more comfort to further development process and provide the competitive advantage over other similar tools. The enhancement goals are the following:

7 - Serialization The serializer & deserializer usable not only for communication between server-side Scala and client-side generated JavaScript.

8 - Loading An equivalent to Java classloader that can load compiled JavaScript files from the server on the fly.

9 - RPC A possibility to mark some methods as remote and infrastructure taking care of the communication.

10 - Application A sample application.

2. Background

Both Scala and JavaScript are not mainstream languages in the sense of chosen paradigms and used concepts so the following sections will focus mainly on the differences from the conventional object-oriented languages (C++, Java, C#) and on differences between the two, which are important from the compiler perspective. If you are already familiar with any of the languages, you may skip the corresponding sections. The section 2.3 is dedicated to the Scala compiler, its internal processes and related data structures.

2.1 JavaScript

JavaScript [3] originated as an in-browser scripting language, more recently it expanded to other areas like game-development, desktop applications and server applications. It was designed by Brendan Eich for Netscape, but when it became adopted by other companies, Netscape submitted the language specification to Ecma international [27] which subsequently formalized it in the ECMAScript language standard [4]. Regarding syntax and control structures, it was mostly influenced by C and Java but from the design point of view, the main principles are taken from Self [28].

Style The language is imperative and to some extent functional. It supports standard control structures (e.g. `if-then-else`, `while`), makes distinction between expressions and statements, the functions are first-class citizens (i.e. objects themselves that can be manipulated with). One big difference is scoping. Unlike most of the standard languages which have block scoping, JavaScript has lexical scoping. So for example a variable declared inside `else` branch of an `if` statement is visible everywhere in the function body.

Type System JavaScript is dynamically typed, types are associated with values, not variables. The possible types are `String`, `Boolean` and `Number` for primitives, `Array` for mutable arrays that behave more like hash tables, `Function` representing functions and methods and `Object` which is the supertype of all of the aforementioned types. Objects are basically associative arrays, so accessing properties is equivalent to dereferencing values in an associative array by property names. Note that there is not any integral numeric type (e.g. `int`, `short`), the `Number` is a 64bit double-precision floating point type. A character type (`char`) is not present either.

Inheritance The most unusual feature is prototype based inheritance whose principles will be demonstrated on an example of hierarchy `Dog` <: `Animal` <: `Object` (the "<:" symbol stands for the "subtype" relation) depicted on figure 2.1.

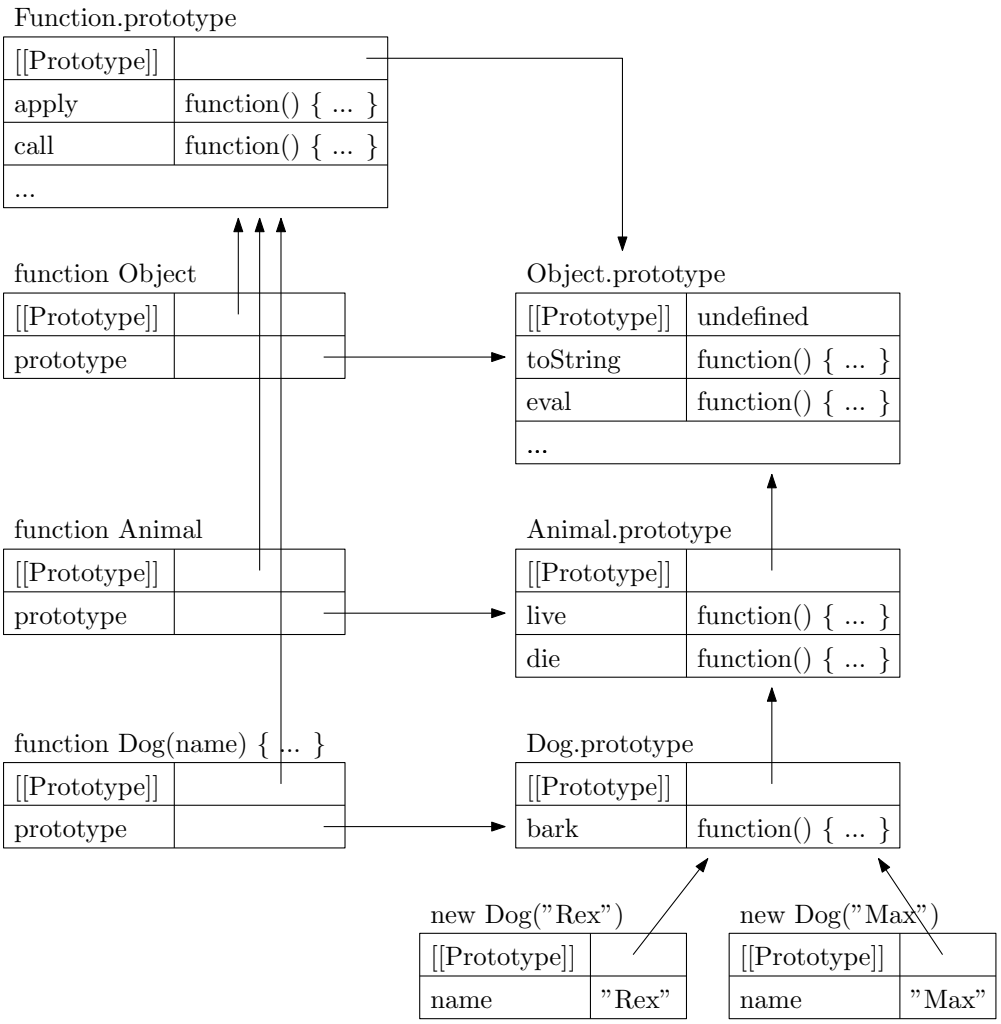


Figure 2.1: JavaScript prototypes, constructors and instances.

The first thing to notice is that all objects (and also functions because functions are objects) have the `[[Prototype]]` property which holds their prototypes. When a property of an object is being accessed, direct properties of the object are inspected first. If there is no direct property with the specified name, properties of the prototype are inspected. And so on until the prototype of currently inspected object is undefined. As a consequence, it behaves like if the object inherited all properties of its prototype. The `[[Prototype]]` property is internal, does not change, is assigned once when the object is constructed and cannot be changed programmatically.

So for example all functions inherit methods `apply` and `call` from their common prototype `Function.prototype` and transitively methods `toString`, `eval` etc. from their prototype's prototype `Object.prototype`.

In order to create objects, a constructor function has to be defined first (e.g. the function `Animal` for animals). A constructor function may access the object that is being created via the `this` keyword and set some of its properties (e.g. the constructor function `Dog` sets `name` of a new dog). Moreover it may have associated special object in the property `prototype` that will be assigned as the `[[Prototype]]` to all objects created with the constructor

function. If the function does not have the `prototype` specified, the standard `Object.prototype` is assigned to the created objects. Finally a new object is created by calling a constructor function together with the `new` keyword, e.g. `new Animal()`.

To sum it up, the hierarchy in the figure 2.1 is set up in the following manner:

1. Define a constructor function for animals - the function `Animal`.
2. Assign a new object with properties `live` and `die` to the `prototype` property of the `Animal` function.
3. Define the dog constructor function `Dog` with body: `this.name = name;`.
4. Create a new `Animal` object (using `new Animal()`), define its method `bark` and set it to the to the `prototype` property of the `Dog` function.
5. Create the dogs using `new Dog("Rex")` and `new Dog("Max")`.

Environment JavaScript is an interpreted language, therefore it requires a runtime environment that interprets it and provides objects and functions for interaction with the "outside world". In the context of web applications, both the runtime environment and the outside world is the web browser. The code is executed in a single thread which behaves similarly to an event loop. There is a queue of code that has to be executed. Code is enqueued to the queue from two main reasons: when a new script is loaded and when an external operation handled by the browser finishes and a callback has to be invoked. The environment cyclically waits until the queue is non-empty and then sequentially executes everything in the queue. So it is guaranteed that nothing will run in parallel.

2.2 Scala

One of the best summaries of the Scala language is provided by its author himself, Martin Odersky, in [29]:

> "Scala fuses object-oriented and functional programming in a statically typed language. Conceptually, it builds on a Java-like core, even though its syntax differs. To this foundation, several extensions are added. From the object-oriented tradition comes a uniform object model, where every value is an object and every operation is a method invocation. From the functional tradition come the ideas that functions are First-class values, and that some objects can be decomposed using pattern matching. Both traditions are merged in the conception of a novel type system, where classes can be nested, classes can be aggregated using mixin composition, and where types are class members which can be either concrete or abstract."

Style The language is both imperative and functional but tends and encourages to use rather the functional style. It supports most of the standard control

structures known from e.g. Java. Adds some new control structures, the most important is the `match` statement (a self explaining example using match is in Listing 2.1).

```
list match {
   case Nil => "empty"
   case 'a' :: tail => "starts with 'a'"
   case (x: Int) :: _ if x > 3 => "starts with int > 3"
   case (x: Int) :: _ => "starts with int"
   case _ => "whatever"
}
```

Listing 2.1: Match statement example.

Scala also generalizes existing control structures where it is possible: `for` cycles are abstracted into much more powerful `for` comprehensions. The `for` comprehensions are actually a syntax sugar for series of operations `map`, `flatMap` and `withFilter` whose semantics are defined by the target of their invocation, not by the `for` comprehension itself. Parallel to that can be found in C# in the form of LINQ [30] which behaves more less the same. An example of one `for` comprehension that selects older underpaid employees can be seen on listing 2.2.

```
for(e <- employees;
        if e.age > 40;
    c <- companies;
        if c.name == e.companyName;
        difference = e.salary - c.avgSalary;
        if difference < 0
) yield (e.name, c.name, difference)
```

Listing 2.2: For comprehension example.

Leitmotif of the Scala language design is unification of concepts which appears for example in the `catch` blocks. Multiple catch blocks for each catched exception type known from standard languages are turned into one that matches against the catched exception. A simple example is provided in Listing 2.3.

```
try {
    throwsException()
} catch {
    case _: TimeoutException => println("timeout")
    case _: IOException => println("IO")
    case e => println(e.getMessage)
}
```

Listing 2.3: Catch block example.

Scala does not differentiate between expressions and statements, because everything that has been traditionally viewed as a statement (`if-else`, `try-catch`) has a return value. So for example a ternary operator is unnecessary, because `if-else` can be used instead (listing 2.4). There are many more features, for a thorough description refer to the Scala "bible" [31].

```
val result = if (a >= 0) "positive" else "negative"
```

Listing 2.4: `If-else` as a ternary operator.

Syntax As it can be seen on the previous code samples, the syntax resembles Java or C#, but is much more flexible while trying to reduce unnecessary clutter. It is possible to define methods whose usage looks like built-in control structures. For example the `using` syntactical construct known from C# can be defined in Scala as a library function whose usage will look exactly the same as in C#. Leaving out parentheses or semicolons is also allowed in places where they are not necessary. As a consequence Scala is pretty suitable for design of domain specific languages (DSLs).

Type System Unlike Java with distinct primitive and reference types, where the primitive types do not belong into the inheritance hierarchy, all Scala types inherit from the common ancestor `scala.Any`. The whole hierarchy is depicted in Figure 2.2, the types that are passed by value are descendants of the `scala.AnyVal`, reference types inherit from the `scala.AnyRef`. The `scala.AnyRef` stands for the standard object of the underlying platform, which does not necessarily need to be Java Virtual Machine but also .NET Common Language Runtime. The only non-standard value type is the `scala.Unit` which represents the product of no types, i.e. the 0-tuple. It has one instance written as empty parentheses - `()`. The `scala.Unit` is for example used as a return type of functions that are in other languages defined as `void`.

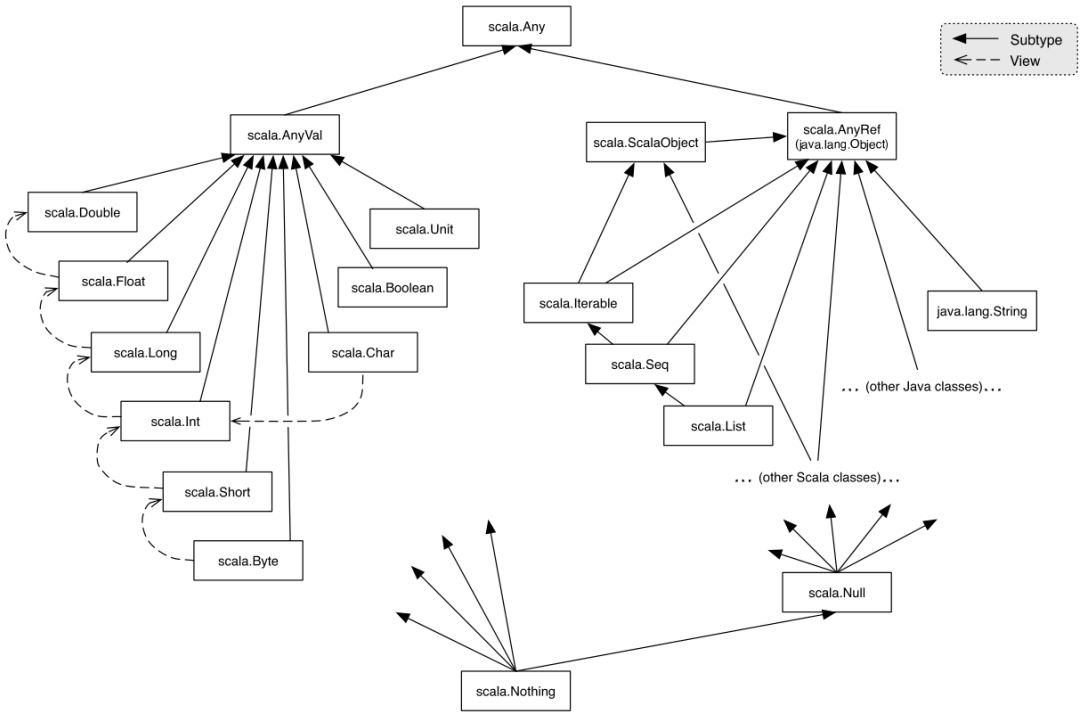


Figure 2.2: Scala Type System.

Note the `scala.Null` type with its only instance `null` and the uninhabited type `scala.Nothing` (i.e. no value has such type, in type theory known as the zero type). With them, it is always feasible to determine common supertype and common subtype for each two types[1].

The types may take other types as parameters (known as generic types), the type parameters can be annotated with variance annotations (invariant, covariant, contravariant) and bounded (e.g. the type parameter has to be subtype of type X). Also some more advanced possibilities are supported - existential types, path-dependent types, structural types, etc (a complete listing can be found at [32]).

Functional Programming Scala also supports a lot of features that can be found only in functional languages so it is possible to write Scala programs solely in the functional style. But not purely because mutable variables which violate the referential transparency are allowed. The most important functional concepts are:

- Closures - anonymous functions that capture the declaration site scope.
- Immutable variables (`vals`).
- Lazy evaluation of variables, function parameters or object fields.
- Higher-order and nested functions.
- Currying and partial application.
- Pattern matching.
- Tuples and algebraic data types.

Object Oriented Programming On the other hand Scala is a pure OO language, meaning that every value is an object. A data type may be declared either as a class, a trait or a singleton object. Classes are quite similar to Java classes, but on top of class inheritance they may mix in any amount of traits. Traits are basically enhanced Java interfaces with a possibility to also define the implementation. From the Scala perspective the only difference between traits and classes is that traits cannot have parametrized constructors.

The `static` fields and methods known from mainstream languages are not supported because it is difficult to abstract over them. To fill the gap, singleton objects behave like classes whose only instance is created during the class-loading process. They serve the purpose of static classes but may also inherit or mix-in other types and possibly override something (this is not achievable in languages that use the `static` modifier).

The famous diamond inheritance problem that always emerges with the multiple inheritance is solved using a technique called linearization [33]. Linearization is a deterministic process that specifies a single linear order for all of the type

[1] A common supertype is needed when inferring type of an expression that returns either type A or type B, a common subtype when unifying two generic types with type parameters on a contravariant position.

ancestors (i.e. all classes in the superclass chain and parent chains of all traits). Method resolution takes advantage of it and works in the linearization order, so there is no place for ambiguity left.

2.3 Scala Compiler

In the context of the thesis it is also important to know how the Scala compiler works, because it may be used by the Swat as a library for tasks that are specific for Scala compilation, but not that relevant from the Scala to JavaScript compilation perspective. The compiler architecture is summarized by Martin Odersky in [29]:

> "The Scala compiler, `scalac`, consists of several phases. The first phase is syntax analysis, implemented by a scanner and a conventional recursive descent parser. The result of this phase is an abstract syntax tree. The next phase attributes the syntax tree with symbol and type information. This is followed by a number of phases that transform the syntax tree. Most transformations replace some high-level Scala-specific constructs with lower-level constructs that can more directly be represented in bytecode. Other transformations perform optimizations such as inlining or tail call elimination. Transformations always consume and produce attributed trees."

So the compiler can be understood as a sequence of phases where each phase takes an abstract syntax tree (AST) as its input from the previous phase, adds attributes to the AST or modifies the AST and passes the modified tree to the consequent phase.

In order to represent a Scala program in the compiler, a couple of data structures needs to be defined (complete description in[34]). `Names` represent identifiers that appear in the source code, e.g. class names or field names. `Symbols` are used to bind a name and the entity it refers to, e.g. a class or a method. Anything that can be given a name has a symbol associated with it. Instances of the `Type` class represent information about the type of a corresponding symbol (the information include member methods, fields, base types etc). Finally, the ASTs are represented by the `Tree` class and its subclasses which correspond to various source code elements such as class definitions (`ClassDef`), method invocations (`Apply`) or `if-else` statements (`If`).

The only up-to-date description of the scalac phases can be found in the source code (class `scala.tools.nsc.Global`). Here are the most important ones in their execution order:

`syntaxAnalyzer`	Takes Scala code as its input, parses source code into ASTs, performs simple desugaring.
`namerFactory`	Declares symbols and enters them into scopes
`typerFactory`	Assigns types to symbols, the most complex phase.

patmat	Converts the `match` expressions into if-else statements, labels and jumps.
uncurry	Performs the opposite of currying, transforms variadic parameters to sequences, converts closures to anonymous classes and a lot of other similar, rather simple, operations.
explicitOuter	For each nested class, adds a field for the reference to the outer class instance.
erasure	Erases generic types, adds interfaces for traits.
lambdaLift	Move nested function definitions to the top level.
lazyVals	Converts lazy fields into methods.
constructors	Moves field initialization into constructors.
mixer	Performs the mix-in composition.
cleanup	Platform specific cleanup.
deadCode	One of many optimization phases, eliminates dead code.
jvm	Generates the Java bytecode.

3. Analysis

This chapter proposes how some of the defined goals can be fulfilled, while providing more possibilities with their pros and cons. It is intended not be biased by the fact that the source language is Scala for the described methods to be applicable even in a different environment. But for example purposes Scala will be used, because it is the source language of the Swat. Moreover this chapter elaborates on which of the proposed approaches is used by the Swat and why.

3.1 Setup

Compiler architecture has converged throughout years to a data flow consisting of canonical components. The source code is consumed by a scanner that performs lexical analysis and produces a stream of lexical tokens. A parser takes the token stream on the input, performs syntactical analysis and builds up the ASTs. The ASTs are then processed by a semantical analyzer which validates them, assigns types to symbols and possibly transforms the ASTs. These three components are marked as the frontend components. When the abstract syntax trees leave the frontend, they go to a backend, which is responsible for generation of the target language/assembly/byte code. The backend usually consists of several subcomponents that firstly produce some intermediate code, optimize it and then generate the machine specific code. Such a data flow can be seen in Figure 3.1.

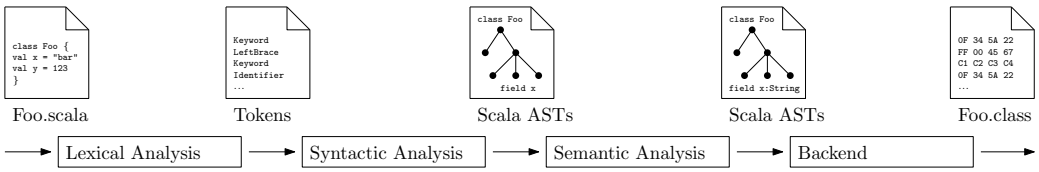


Figure 3.1: Canonical compiler architecture.

In order to transform programs to JavaScript, one needs to work with them programmatically. ASTs exactly serve this purpose and because standard compilers (e.g. javac, csc, scalac) already handle the transformation from source code to ASTs, reusing that functionality is definitely the best way to go.

As it can be seen in the diagram of the whole setup (Figure 3.2), the compiler to JavaScript itself follows the canonical structure, therefore it consists of a frontend and a backend. The JavaScript Frontend in ideal case consumes ASTs obtained from the source language compiler and produces corresponding JavaScript ASTs. When the JavaScript frontend is done, the execution proceeds to the JavaScript Backend. It generates JavaScript code corresponding to the ASTs and may optimize the output either by itself or by an external tool (e.g. Google Closure compiler [13] which is one of the most popular JavaScript optimization tool).

Another advantage of the proposed setup is that compilation to bytecode and compilation to JavaScript can be performed during one run of the composed compiler. So the lexical analysis, syntactical analysis and possibly type checking

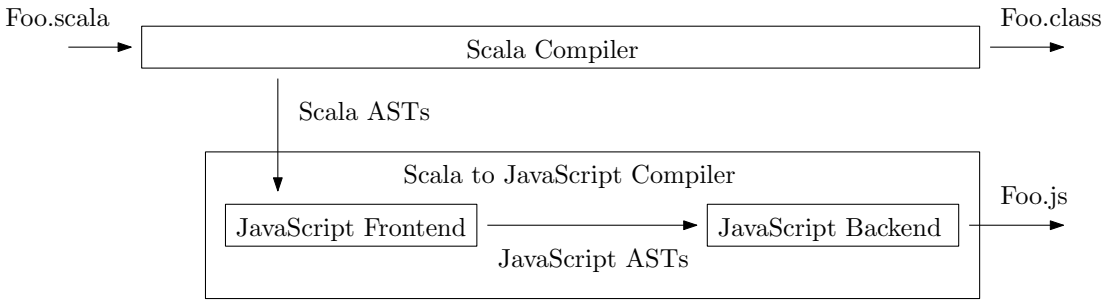


Figure 3.2: JavaScript compiler using the source language compiler.

runs only once and their outcome is used both by the standard compiler and the compiler to JavaScript. And it is a common knowledge that these three operations take the most of the compilation time.

One may also ask why the compilation to bytecode is necessary after all. The first reason is that data structures intended for the client side could be reused on the server side and the other way round. Or at least functionality that is common for the server and the client can be abstracted out in order to avoid duplication. The main reason however is that a program may depend on a library (possibly compiled to JavaScript). If the library did not contain classfiles then the program itself would not be compilable (it would not pass the type checking phase of semantic analysis).

3.1.1 Utilizing the Source Language Compiler

The standard compiler architecture offers a few places where the ASTs can be obtained. As depicted in Figure 3.1, it is either before the semantic analysis is executed (option A) or after execution of the semantic analysis (option C). If the compiler components are subdivided into phases, it may be even possible to access syntax trees during execution of the semantic analysis (option B). That mostly depends on architecture of the source language compiler, whether it allows such an interception.

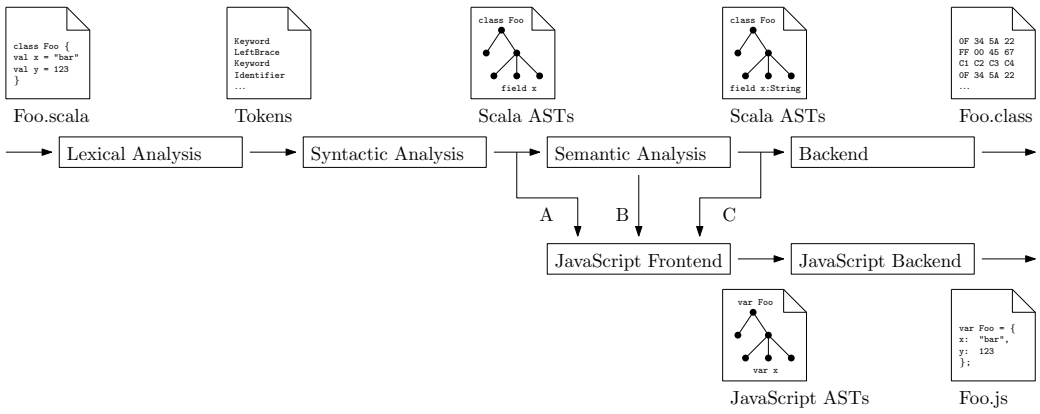


Figure 3.3: Plugging JavaScript compiler into the source language compiler.

The choice between options A, B and C determines abilities of the JavaScript

frontend. The ASTs may just mirror the source code with no added information (option A), they may be completely analyzed and possibly transformed to some extent (option C), or the ASTs may be in an interim state between A and C (option B). Drawback of the option A is that the ASTs may represent language constructs that are not directly expressible in JavaScript. An issue with the option C is that some of the trees may be transformed to constructs that are really low-level and close to bytecode, even though the original trees would be directly expressible in JavaScript before the transformation.

The Scala compiler is subdivided into phases so the most crucial decision is finding the spot where the compiler to JavaScript should be plugged in. Scala compiler phases have fixed order and dependencies and it is almost impossible to leave out a phase from the sequence. So the decision is where to cut the sequence into two parts.

Principally the compiler to JavaScript could be plugged anywhere. The earlier it is plugged in, the more work for it because the early phases work with much richer set of tree types compared to the latter phases where some of the AST types are completely eliminated (e.g. `Match` by phase `patmat`). On the contrary the ASTs that appear on the beginning of the phase sequence practically mirror the original Scala code whereas the ASTs towards the end look more like Java mixed with bytecode.

One of the aims was to produce JavaScript code that would resemble Scala as much as possible and use just the compiler phases that analyze the code (`syntaxAnalyzer`, `namerFactory`, `typerFactory`). The motivation for it was to simplify debugging of the compiled code in a web browser. But it turned out that it would be a lot work to implement JavaScript counterparts of Scala control structures that are not present in JavaScript. So the phases `patmat`, `uncurry` and `explicitOuter` should also be utilized, because they transform higher level Scala constructs to ones that are directly expressible in JavaScript. From the other point of view, the phase `erasure` does not bring any added value in the context of compilation to JavaScript, it erases some of the type information. The phases after `erasure` (namely the `lamdaLift`) transform the trees to an extent, that is not acceptable if the resulting JavaScript code should resemble Scala at least a bit. To sum it up, the place between `explicitOuter` and `erasure` is where the Swat compiler should be plugged in.

3.1.2 Debugging

The chosen plugging position implies that the resulting code sometimes would not resemble the source Scala code which means that debugging of the program directly in JavaScript in a web browser would be tricky. This way of debugging is not ideal even if the outputted JavaScript code looked almost the same as the source it was compiled from. Similar problem appears in the plain JavaScript world where code that went through minification (e.g. whitespace removal, name shortening) is difficult to debug.

JavaScript ecosystem and web browsers offer a solution in form of source maps

[35] that establish mapping between the executed JavaScript code (minified or compiled) and the original code (not minified or even possibly not JavaScript) and allow a user to debug the original code. Source maps are still a state of the art functionality not supported by all browsers and establishing such a mapping can be quite a complex task. Therefore implementation of it is out of the scope of the thesis.

3.2 Interoperability with JavaScript

One of the crucial requirements of a compiler targeting JavaScript is enabling the programs written in a source language to interoperate with the JavaScript environment. There are two main reasons:

- Programs do not run in a vacuum and the only possibility how they can manifest themselves is via interaction with the hosting environment. It is feasible to make them interoperate with the environment in a type safe manner, since the JavaSrcipt and browser APIs are defined mostly in W3C [36] specifications which contain type information. It is JavaScript where the type information is lost.
- Sometimes it makes sense to descend to the JavaScript level and write a piece of code directly in it, for example for performance reasons.

Inherent restriction of the interoperability is that the solution used should not violate the syntax of the source language. There are several ways how to accomplish the proposed requirements, the following three solutions represent different ways of doing it and are more less orthogonal to each other:

Native code blocks A parallel to the inline assembler blocks in the C language with difference that the JavaScript code has to be placed in a string in order not to violate the source language syntax. Moreover type of the whole code block expression (e.g. `js("123 + 456")`) has to be `scala.Nothing` so it can be used even in places where a value of particular type is expected. When the JavaScript Frontend encounters such a block it outputs only the code stored in the string. Example of such a code block can be seen in Listing 3.1.

```
def repeat(text: String, times: Int): String = js("""
  var a = [];
  while(a.length < times) {
    a.push(text);
  }
  return a.join("");
""")
```

Listing 3.1: Native code block example.

Adapters For all the JavaScript APIs and objects that should be usable from the source language, so called "adapter" classes could be defined in the source lan-

guage. They should exactly match the JavaScript definitions in terms of names, fields, methods, parameters etc. And they should be marked with an annotation in order to be distinguishable. They do not have to be implemented, but the type signatures should match the specification (see Listing 3.2 where an adapter for the `window` object is defined and used). When the compiler encounters such an adapter class, it knows that it is something that is already present in the JavaScript environment and will not produce any JavaScript code corresponding to the adapter definition (Listing 3.3 presents such a situation).

```
@adapter object window {
  def alert(text: String) {}
}

def test() {
  window.alert("test")
}
```

Listing 3.2: Adapter definition and usage example.

```
var test = function() {
  window.alert("test");
}
```

Listing 3.3: Adapter compilation output.

Dynamic This option is relevant only to source languages that support the concept of a dynamic class. In Scala, an object named `global` representing the JavaScript global scope (`object global extends Dynamic`) would be defined. The object would mix-in the `scala.Dynamic` [37] trait, so an arbitrary field can be selected or any method can be invoked on that particular object. For example `global.window` would be turned by the Scala compiler before the typing phase into `global.selectDynamic("window")`. Such a selection can later be transformed by the JavaScript Frontend into `window`.

As it was mentioned in the motivation in Chapter 1.1, static typing is an advantage in the context of large applications, therefore choosing *Adapters* is preferrable. Unlike the two other it preserves type safety. The trade-off is that it is incomparably more time-consuming for a developer to define all the adapters compared to the other options where nothing has to be declared. For exceptional purposes (performance optimization) the *Native code blocks*, which are more flexible than *Dynamic* could be used.

3.3 Frontend

A heart of the whole compiler is the JavaScript frontend responsible for transformation of source language ASTs to JavaScript ASTs. It starts given the whole program in a form of a syntax tree, traverses the AST and produces a JavaScript AST. According to the current subtree whose root the fronted is visiting, it should produce a JavaScript counterpart for that subtree. The following sections describe options the fronted has and elaborate on how particular tree types should be represented in JavaScript. It is natural to start with the top-level node that

encapsulates everything else and traverse down the tree through class definitions, their member definitions to the simplest expressions on the leaf level of the tree.

3.3.1 Compile Time vs. Runtime Approach

The JavaScript frontend can use different approaches to tackle its task. As an example, consider a piece of Scala code containing lazy field declaration and access in Listing 3.4 and how it could be compiled to JavaScript.

```
lazy val foo = reallyLongOperation()

// Causes evaluation of foo.
val bar = foo
```

Listing 3.4: Lazy field declaration and access.

One option is to construct JavaScript ASTs utilizing only JavaScript language primitives and internal functions so the resulting code would be executable as it is. Everything is handled during the compilation (thus named the "compile time approach") so the produced code uses only native JavaScript constructs and APIs (listing 3.5).

```
var foo_value = undefined;
var foo = function() {
  if (typeof foo_value === "undefined") {
    foo_value = reallyLongOperation();
  }
  return foo_value;
};

var bar = foo();
```

Listing 3.5: Compile time option outcome.

The other way is to move some of the logic to a completely new runtime library written directly in JavaScript so the outputted JavaScript code would run only if the runtime library is present in the global scope. In case of the lazy field a new function named `runtime.lazy` would be defined in the runtime library (Listing 3.6 shows its implementation).

```
runtime.lazy = function(f) {
  var x;
  return function() {
    if (typeof x === "undefined") {
      x = f.apply(this, arguments);
    }
    return x;
  };
};
```

Listing 3.6: Runtime function that lazifies evaluation.

With the library function `runtime.lazy` it is possible to translate Scala code in Listing 3.4 to JavaScript code listed in 3.7.

```
var foo = runtime.lazy(function() {
  return reallyLongOperation();
});

var bar = foo();
```

Listing 3.7: Runtime option outcome.

In order to minimize size and boilerplate code in the produced JavaScript and to increase its readability, the Swat JavaScript fronted sticks to the runtime option with a runtime library and tries to move functionality there if it is both possible and beneficial. In cases when the compile time option nor the runtime option is apparently better than the other, the runtime option is chosen because modifying the runtime is by far simpler than modifying the compiler.

3.3.2 Scope of a Compilation Unit

The top level tree type represents a compilation unit (one source file) and it contains possibly nested definitions of packages. The package definitions are usually non-empty, they encapsulate type definitions (definitions of classes, singleton objects and traits). An important decision on this level is how the compiler output should be structured. Should it be a single file containing everything in the compiled program/library? Should it be a set of files that correspond to the compilation units? Or should the compilation unit output be splitted into mutliple files? In the web environment such questions have to be answered because size of files downloaded to the client really matters (particulary on mobile devices).

It is a crucial decision because the compiler to JavaScript should be besides other things used to compile the Scala standard library [38] (which is an implicit dependency of all Scala programs). And the Scala library contains thousands of classes so having the library packed in a single file with size exceeding megabytes is not acceptable in any real world scenario.

An entry point to a Scala application is a singleton object that mixes in the `scala.App` trait (an analogue to a class with `static void main` method in

other languages). Not all of web applications are in the form of a single page, it is quite common that there are multiple pages with completely different functionality. The application therefore has multiple entry points. There is a shared portion of the codebase but different entry points have different implementation that is not shared, so it a is a waste of resources to download everything in the context of a single page.

An inspiration for a solution can be found right inside the standard Scala compiler. It splits the compilation unit into class definitions and produces separate classfiles (`*.class`) for each class definition. For a compiler targetting JavaScript, the splitting opens up a possibility, to track and store dependencies of each type definition together with its outputted JavaScript typefile (analogue to a Java classfile). A dependency is either an instantiated class, an accessed singleton object, a used type or a supertype of the type definition. With this information, the JavaScript typefiles form an oriented dependency graph. It would also be possible to track dependencies among outputs of compilation units, however it would not be any advantage over dependencies among type definitions.

In order to run an application on a page, a typefile corresponding to the entry point has to be loaded with all its transitive dependencies. Utilizing the dependency graph ensures that only the types that are actually used are downloaded to the client. The same applies for external libraries. If an application uses just a fraction of an external library, then the downloaded JavaScript file still stays relatively small.

3.3.3 Type Definitions

Descending an AST, the next kind of nodes JavaSript frontend comes accross are type definitions. Packages or namespaces are not an issue because solution for them is trivial as it is illustrated in Listing 3.8. In order to declare a package it is enough to create an object with the same name as the top level package and recursively nest there objects corresponding to the subpackages.

```
// Declaration of the package "foo.bar.baz".
var foo = { bar: { baz: {} } };

// Definition of a package member.
foo.bar.baz.Animal = /* ... */
```

Listing 3.8: Packages in JavaScript.

A type definition (a class, a trait or a singleton object definition) is identified by a name, has an ordered sequence of supertypes, consists of members and has a constructor. A member is either a method definition or a field definition. In the case of a singleton object or static constructors, the runtime is responsible for their initialization.

There are many possibilities, how to encode such type definitions in JavaScript, because JavaScript is really flexible. Different JavaScript application frameworks,

that introduce the concept of class-based inheritance, do not even agree and encode it slightly differently. As mentioned earlier, the favored encoding would be to produce only necessary information about the type definition. It means that a function (e.g. `runtime.class`) has to be defined in the runtime library. This function should take three parameters: fully qualified name of the class, class supertypes and the class member definitions. Its responsibility is to create a constructor function for the class and setup inheritance by initializing the prototype chain.

An example of how a Scala class `Dog` (Listing 3.9) could be encoded is visible in Listing 3.10. The `runtime.class` function in this case returns a contructor function of the `animals.Dog` so new instances can be created simply by calling `new animals.Dog()`.

```
package animals {
  class Dog extends Animal with FourLegged {
    def bark() { /* ... */ }
    override def live() { /* ... */ }
  }
}
```

Listing 3.9: Scala `Dog` class example.

```
animals.Dog = runtime.class(
  "animals.Dog", // Full name of the class.
  [Animal, FourLegged, java.lang.Object, Any], // Super types.
  {
    bark: function() { /* ... */ }, // Member function definition.
    live: function() { /* ... */ }  // Member function definition.
  }
);
```

Listing 3.10: The `Dog` class encoded using the `runtime.class` function

One issue that has to be taken special care of is an initialization of singleton objects. Singleton objects behave similarly to static constructors, they should be initialized before they are accessed for the first time. To ensure that it is sufficient enough not to reference them directly, but use an accessor function instead which follows the lazy initialization pattern [39]. When a singleton object is accessed for the first time via the accessor function, the function knows it has not been initialized yet, initializes it and returns in.

3.3.4 Scope of a Type

A type has two kinds of members: field definitions and method definitions. Fields of JavaScript objects do not have to be declared in advance, they come into existence when they are assigned. Therefore the JavaScript frontend can safely ignore the fields that are only declared but not initialzied. The assignment part (e.g. `foo = 123`) of fields that are both declared and initialized together (e.g.

`val foo = 123`) has to be moved into a new synthesized field initialization function. The field initializer invocation has to be put in the beginning of each constructor so when a constructor body is being executed, the fields are already initialized.

In Scala there is notion of a default constructor. If a class has multiple constructors, the overloaded constructors have to, as their first statement, call either another overloaded constructor or the default constructor. But the invocation chain always has to end up in the default constructor. Otherwise the Scala compiler reports an error. As a consequence there is no need for the synthetic field initializer, the field initialization can be placed on the top of the default constructor body.

Scala also supports lazy fields that are initialized when firstly accessed. The same approach as with the singleton object initializers can be utilized which gives an opportunity to define a runtime function `runtime.lazy` (implementation can be seen in Listing 3.6) usable for both cases.

The situation is much more complicated with method definitions. Depending on the source language, a method can be overloaded and it can have parameters with defualt values or a variadic parameter. Some of the features may already be handled by the source language compiler, namely the variadic parameters (but even if not, their processing is not a big challenge).

Method overloading

Because JavaScript does not support multiple variables with the same name in a scope, methods of JavaScript objects cannot be overloaded. Listing 3.11 presents a pair of overloaded methods and their invocation.

```
def foo(x: String) { /* ... */ }
def foo(x: Int) { /* ... */ }
def test() {
  foo("test")
  foo(123)
}
```

Listing 3.11: Scala method overloading example.

Overloaded methods differ by their type singatures, therefore one possible solution is to somehow encode the type signature into the method name (known as name mangling which is a commonly known technique that is used by C++ compilers). Output of a compiler that uses name mangling applied on the sample methods can be seen on in Listing 3.12.

```
foo__java_lang_String = function(x) { /* ... */ };
foo__scala_Int = function(x) { /* ... */ };
test = function() {
  foo__java_lang_String("test");
  foo__scala_Int(123);
};
```

Listing 3.12: Overloading solved by name mangling.

The other way is to have one "wrapper" method named the same as all the overloaded methods. The wrapper method would take a type signature as an additional string parameter and based on its value, it would decide which variant to actually invoke. This approach implies that a runtime library function has to be defined. It can be named e.g. `runtime.overload`, as parameters it would take the overloaded variants with their type signatures and it would construct the "wrapper" function. This technique applied on the sample methods is presented in Listing 3.13.

```
foo = runtime.overloaded({
  "java.lang.String": function(x) { /* ... */ },
  "scala.Int": function(x) { /* ... */ }
});
test = function() {
  foo("test", "java.lang.String");
  foo(123, "scala.Int");
};
```

Listing 3.13: Overloading in JavaScript.

The "name encoding" approach has a big advantage in zero runtime overhead. On the other hand, methods with a few parameters of nontrivial types would have really long names so the resulting code would be rather unreadable. The most important information about a method call in the use site is the method name and the arguments. With this solution, they would be separated by the type signature encoding.

The "wrapper" way obviously causes runtime overhead both during declaration (the `runtime.overload` function has to create the "wrapper" method) and during invocation (choosing the variant according to the provided type signature). A benefit is that the resulting code is more readable - the important information occur in the code earlier than the less important type signature. Another advantage is that calling a method compiled to JavaScript directly from JavaScript is simpler, because the one who calls it does not have to know, how a type signature is encoded into a method name. It is enough to know fully qualified names of the method parameter types.

If performance is the main priority, then the "name encoding" variant is the one to use. In Swat, the "wrapper method" approach is used mainly for the sake of code readability, however it would not be a big effort to change it to the other one.

Default Parameter Values

If the source language allows to define method parameter default values, then the best approach is to exploit the JavaScript `undefined` constant which has no counterpart in most of other programming languages. In the invocation site if an argument is not provided (i.e. the default value should be used) the JavaScript frontend can use the `undefined` as the argument. And on the top of the method body, all the arguments have to be checked whether they are the `undefined`. If so, the default value should be used instead as it is demonstrated in Listing 3.14.

```
function(x) {
  if (typeof x === "undefined") {
    x = "default value";
  }
  // ...
}
```

Listing 3.14: Default parameters in JavaScript.

3.3.5 Expressions

The last category of ASTs are expressions. An expression is basically anything that can appear inside a method body (except type and function definitions). The most common expressions are control structures, method invocations, field selections and literals.

Scoping

The first issue that has to be resolved is different scoping in JavaScript. Scala and most of the other standard languages use block scoping where a variable is availible only in the block it is declared in. JavaScript on the contrary uses lexical function-level scoping therefore a variable is available in a function body where it was declared in. The difference in scoping is illustrated in Listings 3.15 and 3.16.

```
val x = "foo"
{
  val x = "bar"
}
println(x) // Prints "foo".
```

Listing 3.15: Scala scoping.

```
var x = "foo";
{
  var x = "bar";
}
console.log(x); // Prints "bar".
```

Listing 3.16: JavaScript scoping.

A solution for this problem is a part of common knowledge in the JavaScript community. The code block that should be executed with a new scope has to be wrapped inside an anonymous function which is immediately executed (see Listing 3.17 for an example). Invocation of the function forces creation of a new scope so it behaves the same way as blocks in standard languages.

```
var x = "foo";
(function(){
  var x = "bar";
}());
console.log(x); // Prints "foo".
```

Listing 3.17: Emulation of block scope in JavaScript.

Another problem related to scoping is meaning of the `this` keyword. In OO languages it means the current object whereas in JavaScript it references the current scope. An issue arises inside a method body if the current object is being accessed from a nested function. The `this` keyword inside the nested function references the nested function scope, not the current object. Resolution of this problem is surprisingly simple. It suffices to capture the current object to a new variable on the top of each method body (e.g. `var self = this;`) and to use the captured variable (`self`) instead of the `this` keyword.

Primitive Types

Due to the fact that JavaScript primitive types differ from primitive types of mainstream OO languages, it has to be decided how the source language primitive values would be represented in JavaScript. The `scala.Boolean` and `java.lang.String` can be represented directly as their JavaScript counterparts, the `scala.Unit` as the `undefined`. But what with the rest?

One possibility is to use the existing JavaScript primitive types which would mean that a `scala.Char` would be represented as a single-character `String` in JavaScript and all numeric values (of types `scala.Int`, `scala.Double` etc.) would be represented as JavaScript `Numbers`. An advantage is that there is no runtime overhead, that the resulting code is well readable and that interoperability with JavaScript is problem-free. From the drawback point of view there is no way how to determine whether `2.0` represents a `scala.Integer` or a `scala.Double`. Moreover overflows do not behave as expected because most of the numeric types have wider ranges than in the source language.

The other way is to create a wrapper class for each type. The class would contain the value represented same way as it is described in the previous paragraph. On top of that it will define all operations (e.g. `add`, `subtract`, `multiply`, etc.) so that overflows and ranges would work exactly the same as in the source language. A benefit is that no information is lost when compiled and executed in JavaScript. A disadvantage is that there is some runtime overhead and that the produced code is little more literal (e.g. `1 + 1` would compile to `new scala.Int(1).add(new scala.Int(1))`). In addition the wrapped values would have to be unwrapped on the boundaries between compiled JavaScript code and native JavaScript code.

The choice is biased by the fact that in most cases JavaScript applications do not not perform any computationally complex operations where bit arithmetic is involved. They however interoperate a lot with the environment and external

libraries. While keeping that in mind it is apparent that the first option (without wrappers) has been chosen, even though the "wrapper" approach is definitely cleaner. A compromise could be a parameter of the compiler that would switch between the two aforementioned approaches.

Control Structures

Thanks to Scala compiler phases that run before the Swat compiler it is not necessary to worry about advanced control structures like `match` because they are desugared to code that is "compatible" with JavaScript. The only thing to take care of is that Scala control structures may have a return value. An elegant solution naturally pops up together with the scope handling. Consider an `if-else` control structure on the listing 3.18.

```
val decision =
  if (foo) {
    "foo is true"
  } else {
    "foo is false"
  }
```

Listing 3.18: A condition with return value.

Transformation to JavaScript will be performed in two steps. The first step in Listing 3.19 is just for illustrational purposes. It is not a valid JavaScript code but it shows how scoping would be resolved (both branches wrapped with immediately invoked functions). The second step is to merge the two anonymous functions into one and add the `return` keywords (Listing 3.20).

```
var decision =
  if (foo) {(function() {
    "foo is true";
  }())} else {(function() {
    "foo is false";
  }())}
```

Listing 3.19: The first step of condition compilation.

```
var decision = (function() {
  if (foo) {
    return "foo is true";
  } else {
    return "foo is false";
  }
}());
```

Listing 3.20: The result of condition compilation.

The `super` Keyword

The last challenging expression kind is an invocation of method defined on a supertype, e.g. `super.foo()`. Consider the setup in Listing 3.21 - a top level trait, two child traits and two classes that mix in both child traits.

```
trait A {
  def foo = "A"
}
trait B1 extends A {
  override def foo = "B1 " + super.foo
}
trait B2 extends A {
  override def foo = "B2 " + super.foo
}

class C1 extends B1 with B2 { def test = super.foo }
class C2 extends B2 with B1 { def test = super.foo }

println((new C1).test) // Prints "B2 B1 A".
println((new C2).test) // Prints "B1 B2 A".
```

Listing 3.21: The `super` keyword non-static behavior example.

Note that the traits are mixed into classes in opposite order to each other. Therefore the linearized supertype sequences of `C1` and `C2` are different and the `test` methods do not return same values. As a consequnce the `super.foo` used inside the method `B1.foo` may reference either `B2.foo` or `A.foo`. So when the compiler processes the `B1.foo` method body, the `super.foo` invocation cannot be dispatched statically.

As it started to become a custom, runtime comes to the rescue. A method `runtime.super` has to be defined. It would take a method name, method arguments and most importantly the declaration site type. The `super.foo` inside `B1.foo` would be compiled into `runtime.super("foo", [], B1)`. When the `runtime.super` method is invoked, it basically starts at the object (e.g. `new C1`) and climbs up the prototype chain (e.g. `C1`, `B2`, `B1`, `A`) until it gets to the declaration site type (`B1`). Then the first method defined above the declaration site type with the specified name is invoked (e.g. `A.foo`).

3.4 Standard Libraries

Most of programs in any language depend on a standard library which is offered together with the source language. In some cases the standard library is a required dependency that cannot be left out even if the program did not explicitly use anything from it. So in order to execute compiled programs there also have to be a JavaScript counterpart for the standard library. If the source code of the standard library is publicly available then the best approach is to use the compiler to JavaScript and compile the library with it. If the sources are not publicly available then the only option is to reimplement it either directly in JavaScript or in the source language and then compile it. Applications in Scala implicitly rely on the Scala Library [38] and a small subset of the Java Class Library. Solution in this case would be to compile the opensource Scala Library classes directly to JavaScript. The Java Class Library subset would have to be reimplemented in Scala and then compiled.

One approach is to take the whole library and compile it as it is. The main downside is that a lot of classes that are not usable in the context of web browsers would be included. For example everything related to threading. Scala collections define the `par` method which returns parallelizable version of the collection. So the parallel versions are direct dependencies of the sequential versions and consequently would have to be downloaded whenever a standard sequential version is used.

The better approach is to start with a small subset of the library classes that would be compiled to JavaScript (i.e. supported classes). Before a class would be added to the subset (i.e. become supported) one would have to check its dependencies and possibly tweak its source code in order to exclude everything unnecessary. In a case of the collections it would mean that the `par` method would be removed, which would break the dependency on the parallelizable version.

3.5 Library Distribution

A way how to simply distribute libraries is one of the main goals of the toolkit. A motivation behind it is thoroughly described in Section 1.1. The solution however, to further extent, depends on the source language platform and distribution means there. In the Scala ecosystem, programs and libraries are distributed exactly the same way as in the Java environment - in the form of `*.jar` files (Java Archives). Such an archive usually contains class files, metadata associated with them and additional resources.

Integrating distribution of the compiled JavaScript with the existing methods of Scala program distribution is without a doubt the ideal approach, because the existing infrastructure (build tools, dependency management tools) could be utilized. The proposed solution is, to trivially output all the generated JavaScript files into the resources directory of the very same module. During the standard compilation, which produces classfiles, the compiler to JavaScript is executed in order to produce the JavaScript typefiles. The typefiles are stored into the resources of the module within the same directory structure that is used in the `target` directory for classfiles. Finally, when the module is packaged into a `*.jar`, both the classfiles and typefiles are included. And that is exactly what the user of the library needs. The classfiles for standard compilation and execution of pragrams that use the library, the typefiles for client side programs that are compiled to JavaScript and rely on the library.

Such a distribution method implicitly raises a new requirement on a JavaScript type loader which should be able to extract typefiles from the resources. Based on a fully qualified name of a type it should obtain the corresponding typefile. Because a typefile may have other typefiles as its dependenecies, the typeloader should explore the dependency graph and yield the required file together with all its transitive dependencies. Therefore the requested type could be immediately declared and used in a web page.

4. Implementation

4.1 Environment

The nature of the Swat toolkit and how it should be used by other programmers implies that it has to be tightly cooperating both with the standard Scala compiler and the build tool SBT. The compiler to JavaScript works with internal data structures and interfaces defined by the Scala compiler, however those may change even between two minor versions. No backward compatibility of those internal structures is guaranteed. As of writing the thesis the most current version of Scala (Scala 2.10.1) has been chosen to be the only supported version.

Build process and execution of the toolkit is handled by SBT, which means that the only requirement on the host system is to have the Java Runtime Environment in version 1.6 or higher installed. All dependencies of the Swat (e.g. Scala compiler, Scala library) are obtained by SBT from the internet. SBT is distributed together with the toolkit in order to make the usage as simple as possible - just downloading the toolkit and running the included SBT. The whole toolkit is divided into multiple projects; everything is defined in a SBT configuration file (`project/SwatBuild.scala`). SBT is not tightly connected with any particular IDE, but it is possible to generate IDE specific project files for the most used ones (IntelliJ Idea, Eclipse) based on the SBT configuration file.

4.2 Toolkit Overview

The two core projects are the `api` and `compiler`. They are the only dependencies required to be able to compile Scala programs to JavaScript. The `api` contains adapter classes and Swat specific annotations that are meant to be used by the programs compiled to JavaScript. The `compiler` performs the compilation step and incorporates most of the logic that was described in Section 3.

All the projects that enable execution of the compiled programs are placed in the `runtime` directory and are compiled using the Swat compiler itself. They also contain sources directly in JavaScript which are not affected by the Swat compiler. As it is depicted in Figure 4.1 the `runtime` projects depend on the `api` (in order to interoperate with JavaScript environment), get compiled by Swat compiler and packaged to Java Archives (consisting of Java classfiles and JavaScript typefiles).

The `runtime/java` is a reimplementation of core Java classes in Scala so it is possible to compile them using Swat compiler. The `runtime/scala` is a carefully selected subset of Scala Library classes that is supported by the Swat toolkit. The existing sources of Scala Library are used and modified with respect to fact that the classes will be executed as JavaScript so for example everything related to threading can be excluded.

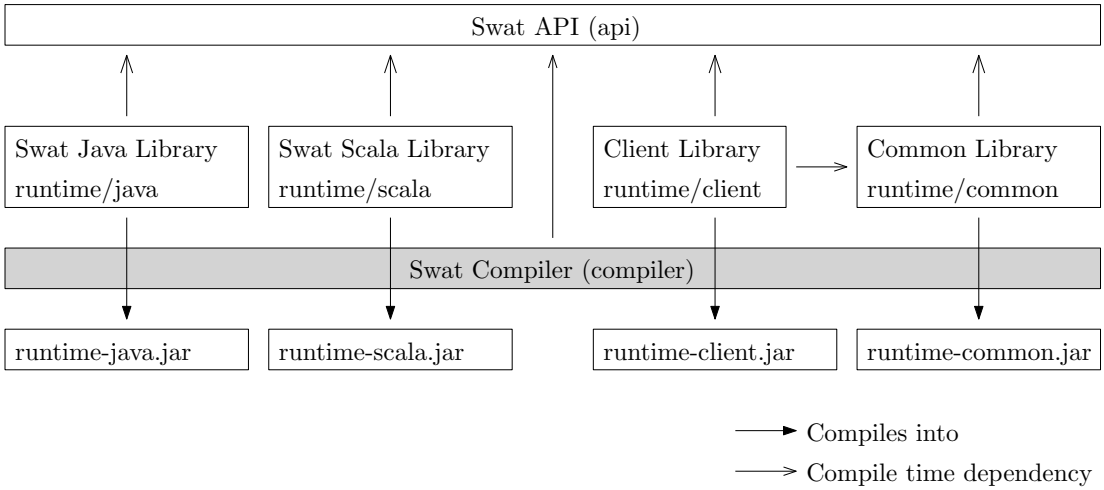


Figure 4.1: Runtime project compilation.

The `runtime/client` is the runtime library that was mentioned multiple times in *Analysis* (section 3). It is partially implemented in Scala and partially directly in JavaScript. Finally the `runtime/common` contains the enhancements like the type loader or the RPC infrastructure. A part of it runs only on the server, part of it is compiled to JavaScript and used both on the server and on the client.

There are four additional projects that either test or demonstrate capabilities of the toolkit. The `web` in the root is a server side web application based on the Play Framework [25]. It serves three web pages where each page contains one client-side application written in Scala and compiled to JavaScript using the Swat compiler. The `web/tests` is an application that executes tests of the Swat Runtime (there is no other simple and direct way of the runtime testing process). The `web/swatter` is a sample application that presents some of the toolkit features. Last but not least, the `web/playground` is an empty application meant to be used for experiments.

An application that should be compiled into JavaScript depends during compilation on the `api` (in order to use adapters), `runtime/client` (so it can use e.g. serializer) and `runtime/common` (for access to type loader etc.). It also implicitly depends (as all Scala applications) on the Standard Scala Library and Java Class Library. It does not depend on the versions `runtime/java` and `runtime/scala` adjusted for Swat. Consequently all types defined in the application could be used even on the server side because they do not depend on the adjusted versions of standard libraries. The purpose of the adjusted versions is to store typefiles of standard Java and Scala library classes. The compilation process is depicted in Figure 4.2.

4.3 API (package `swat.api`)

The `api` library is completely visible to users of the toolkit and contains means for interoperability with JavaScript environment and a few annotations that somehow affect the compilation process. The annotations are currently only two, the

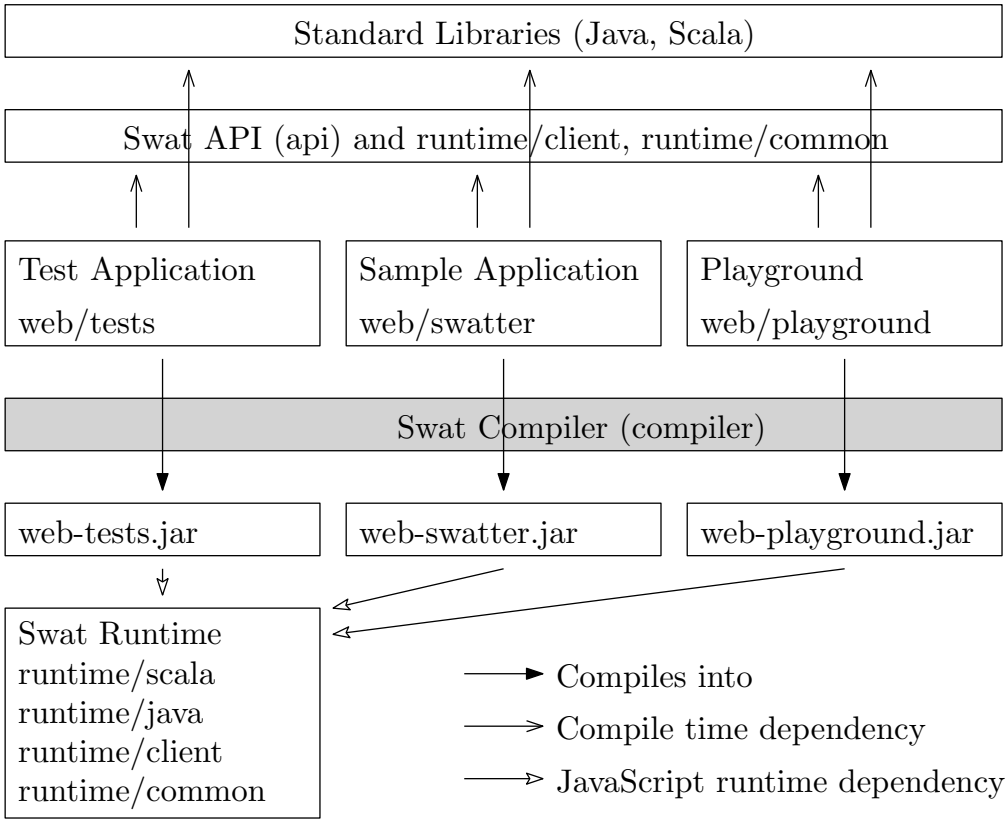


Figure 4.2: Swat application compilation.

`swat.ingored` makes the Swat compiler completely ingore the annotated type, the `swat.adapter` informs the compiler that the annotated type is an adapter for a JavaScript object.

The most important parts of the `api` are predefined adapters for native JavaScript APIs. It is not meant to cover all possible JavaScript APIs, mostly the adapters are added/extended when needed during development of an application or a library. There are couple of predefined adapters that should more less cover the following JavaScript APIs:

- Document Object Model (DOM) Level 3 Core Specification.
  Package `swat.js.dom`.
  `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`
- Document Object Model (DOM) Level 3 Events Specification.
  Package `swat.js.events`.
  `http://www.w3.org/TR/DOM-Level-3-Events/`
- JavaScript Objects, Browser Objects and HTML DOM Element Objects.
  Packages `swat.js`, `swat.js.applications`, `swat.js.html`.
  `http://www.w3schools.com/jsref/default.asp`
- Web Workers and Communication.
  Packages `swat.js.workers`, `swat.js.communication`, `swat.js.url`.
  `http://www.w3.org/TR/workers/`

The first thing to notice is that the adapters do not blindly copy the specifications because the specifications are in some cases redundant and contain duplicities. Usage of Scala, which is great for abstraction of common functionality, allows to define a new trait (with no counterpart in the specification) for the shared operations. The trait can later be mixed into all adapters that have the specified operations. The `swat.js.html.elements.InputLike` trait is an example of such an approach. It defines an interface that is shared among all form input HTML elements and is mixed into `Input`, `Select` and `TextArea` adapters.

The second interesting part is how the global scope is established. The global scope richness depends on a context in which JavaScript code is executed. If a script is being executed in the standard context (applies to all scripts inside a page) then it may access the `window` object or manipulate with the page. On the other hand if a script is being executed inside a web worker (analogue to a parallel process) then it is forbidden to access the `window`. So the object `swat.js.CommonScope` defines API that is availible in all execution contexts, `swat.js.DefaultScope` defines what is availible inside the default execution context and `swat.js.SharedWorkerScope` what is accessible in a web worker. The preferred way is to use the `swat.js.CommonScope` which ensures that the code would work in all execution contexts. A more specific global scope should be used only when it is really necessary.

4.4 Compiler (package `swat.compiler`)

The Swat compiler is implemented as a plugin (`SwatCompilerPlugin`) to the Scala compiler so it follows the Scala compiler plugin design guidelines [40]. The plugin adds one phase to the standard set of compiler phases. When the phase is executed during the compilation it does not modify the input Scala ASTs at all and delegates control to the JavaScript frontend (class `ScalaAstProcessor`) as it can be seen in Figure 4.3. Additionaly, there is also the `SwatCompiler` class which wraps the whole plugged Scala compiler, adds a programmer-friendly interface thus it is possible to perform the compilation to JavaScript programmatically.

There are three main subpackages: `js`, `frontend` and `backend`. The `js` package contains definitions of JavaScript ASTs and other syntax related constructs (e.g. keywords). The classes that represent syntax trees mirror the specification of JavaScript syntax [4] so any imaginable JavaScript program could be represented with these classes. In addition, the `TreeBuilder` is a factory for some commonly used ASTs.

The `backend` package allows to generate JavaScript code from the ASTs. There is only one important class - `JsCodeGenerator`. From the top level point of view, it is just one big `match` expression against all types of ASTs where each `case` is responsible for one type. In most cases it works recursively because an AST is often composed of other ASTs. Therefore the encapsulated ASTs are processed first and then the resulting code fragments are somehow combined together. It also takes care of the indentation and white spaces of the outputted

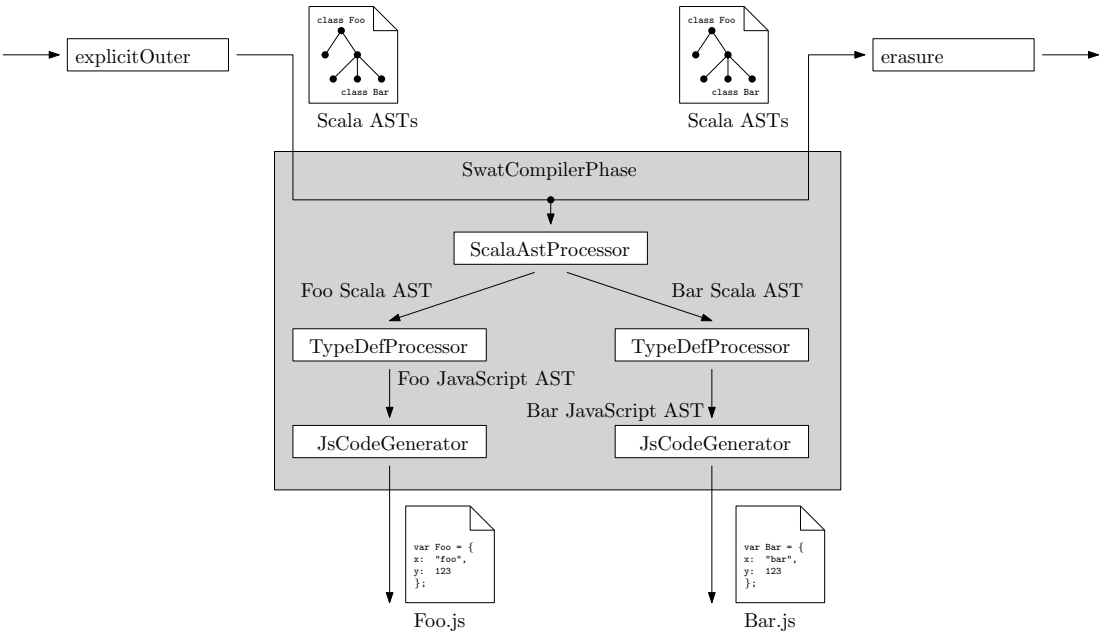


Figure 4.3: Swat compiler implementation.

code to make it readable.

4.4.1 Frontend (package `swat.compiler.frontend`)

An entry point to the frontend is the `ScalaAstProcessor` class. Its functionality is pretty straightforward as it just traverses a body of a `CompilationUnit` in order to find all type definitions. Then for each type definition it creates a new `TypeDefProcessor` and lets it process the type definition.

The `TypeDefProcessor` is an abstract class responsible for a transformation of a type definition Scala AST to a JavaScript AST. It defines a functionality common for all kinds of type definitions and for each kind it has a corresponding subclass: `ClassProcessor` for class definitions, `TraitProcessor` for trait definitions and `ObjectProcessor` for singleton object definitions. The subclasses are very simple, most of the functionality is shared and defined in the `TypeDefProcessor`.

The approach used inside the `TypeDefProcessor` is very similar to how the `JsCodeGenerator` generates JavaScript code, the main difference is that the `TypeDefProcessor` generates JavaScript ASTs. First of all, methods are extracted from the type definition, groupped by their names and processed as it is described in the *Analysis*. Processing a method group means that all overloaded variants are processed separately and then encoded to a single JavaScript AST representing the whole method group. Finally when all the method groups are processed, the type definition can be encoded into a single JavaScript AST containing the members, type name and linearized supertypes.

The transformation on the expression level (e.g. method bodies, field values) starts with the most important method `TypeDefProcessor.processTree`.

The method can be given any kind of AST and based on its type the `processTree` delegates the control to a more specific method, which is responsible for that particular AST. For example when the `processTree` is given an `If` tree it delegates the control to the `processIf` method. On top of that there are several variants of the `processTree`, which invoke it and somehow modify or check the result:

`processStatementTree` Makes sure that the processed JavaScript AST is a `js.Statement`. Otherwise, it tries to convert it to a statement. Unlike the `processTree` method, which returns a generic `js.Ast` it returns a `js.Statement`.

`processExpressionTree` Makes sure that the processed JavaScript AST is a `js.Expression`, which is also the return type. Used for example when processing a condition of an `If` AST. The condition always has to be an expression so `processTree` is too generic for that purpose.

`processReturnTree` Makes sure that the processed AST ends with a `return` statement. E.g. used for transformation of expressions that represent method bodies.

The rest of the `TypeDefProcessor` methods are the aforementioned variants tailored for concrete AST types, e.g. `processLiteral` for primitive literals, `processIdent` for identifiers, `processMatch` for match statements and many more. An important thing to remember is that all of the methods follow the chosen approaches defined in the *Analysis*. From the implementation perspective they are not that interesting to further extent.

4.5 Runtime

4.5.1 Client (package **`swat.client`**)

A core of the client-side runtime is implemented directly in JavaScript and can be found in the resources of the project (file `resource/swat.js`). It is the first thing that has to be declared in order for the compiled code to function. First of all, it defines several helper methods (e.g. `isJsObject`, `isJsArray`) that are not present in JavaScript yet are useful. Then there are functions used by the JavaScript Fronted to encode Scala constructs: `swat.type` encodes classes and traits, `swat.object` encodes singleton objects, `swat.method` encodes possibly overloaded methods. The last category of functions is responsible for operations that are implemented in Scala with respect to the underlying platform (usually JVM):

`swat.isInstanceOf` Returns whether an object is of the specified type. Basically verifies that the object is either of the type or any of the object supertypes <: the specified type.

`swat.asInstanceOf` Uses the `isInstaceOf` to check whether an object is of the specified type. Throws a `ClassCastException` if it is not.

swat.equals Implementation of the == relation (in Scala == corresponds to the `equals` method in mainstream languages, Scala `eq` corresponds to reference equality == in e.g. Java). It cannot be a method of the `scala.Any` type because even `null == null` should return true. For more about equality and its common pitfalls refer to [53].

swat.getClass Returns a `java.lang.Class` of the invocation target. In Swat the `java.lang.Class` is rather simple, the reflection is not currently supported.

swat.hashCode An implementation of the Scala method `hashCode` (or under alias `##`). In order to work properly, each instance of any Swat type is given an identifier, which can be used as a base for its hash code.

swat.toString Stringifies any given object.

4.5.2 Type Loader

The class `swat.commmon.TypeLoader` is not complex at all, it provides two operations: `get` for JavaScript code of a specified type (including all its dependencies) and `getApp` for JavaScript code of an application and its execution (invocation of the application constructor). It works in two phases. In the first phase it creates a dependency graph of all involved types by visiting all dependencies of the specified type, their dependencies and so on. Dependencies can be of two kinds:

Declaration JavaScript code of the dependency has to be declared before the type itself is declared.

Runtime Dependency has to be declared, however order of the dependency declaration and type declaration does not matter.

For example the core runtime `swat.js` is a declaration dependency of any other type. Then in the second phase the graph is traversed in a topological order given by the "declaration" dependency and outputted to one string.

There is also a possibility not to include some types to the result. That is useful when there are types already declared on the client-side and some other have to be loaded. In this case it is unnecessary to download the source code of already declared types again and redeclare them again.

4.5.3 JSON Serializer

Besides serialization of primitive types and arrays, the serializer is also capable of object serialization. In order to support serialization of object graphs that contain cycles, all objects have to be given unique serialization ids and referenced indirectly using the ids. Listing 4.1 shows a self explaining example of how an object (with couple of fields that reference other objects) would be serialized into JSON.

```
{
  $value: { $ref: 0 }, // The root object.
  $objects: [ // Repository of all AnyRef objects.
    {
      $id: 0, // Id of the object, which is used while referencing it.
      $type: "my.package.X", // Type of the object.
      a: "Foo", // AnyVal.
      b: { $ref: 1 }, // Reference to an object.
      c: { $ref: "scala.None", // Reference to a singleton object.
      d: [ 123, 456, 789 ], // Array of AnyVals.
      e: [ { $ref: 0 }, { $ref: 1 }, { $ref: 2 } ] // Array of AnyRefs.
    },
    {
      $id: 1,
      // ...
    },
    {
     $id: 2,
     // ...
    }
  ]
}
```

Listing 4.1: A serialized object that references other objects.

Notice the included type information so it is possible to succesfully deserialize it on the server and assign proper prototype on the client. When a singleton object should be serialized then a fully qualified name of it is used as a reference.

The deserializer on the other hand goes through the JSON, creates instances corresponding to every single object in the `$objects` array while using `null`s instead of the references (the referenced instances may not have been created yet). When all instances are created, the references that were initially set to `null`s can be corrected so they point to valid instances.

4.5.4 RPC

The remote method invocation is a concept that allows to communicate between two physical machines by invocation of a method. The programmer does not need to take care of the communication, serialization of the exchanged data etc. Everything is handled by the underlying infrastructure. In the context of web applications, the client application can directly communicate with one other application - the server application. First of all, the target of invocation has to be defined. Consider a singleton object (`foo`) with one method (`echo`) in Listing 4.2.

```
@remote object foo {
  def echo(message: String): String = {
    "Received message '" + message + "'."
  }
}
```

Listing 4.2: A remote object.

When the `echo` method is invoked from within the client application then the call should be somehow delegated to RPC infrastructure which dispatches it to

the server. A support from the Scala to JavaScript compiler is needed in order to achieve that. Whenever the compiler comes accross a method invocation on an object that is annotated with the `@remote` annotation it delegates the call to the runtime. An example of such invocation can be seen in Listing 4.3 and how it gets compiled to JavaScript in Listing 4.4.

```
def test() {
  foo.echo("Hello World!")
}
```

Listing 4.3: Remote method invocation in Scala.

```
var test = function() {
  swat.invokeRemote(
    "foo.echo",
    ["Hello World!"]
  );
}
```

Listing 4.4: Remote method invocation in JavaScript.

The sequence of steps that follows invocation of the `swat.invokeRemote` is depicted in Figure 4.4.

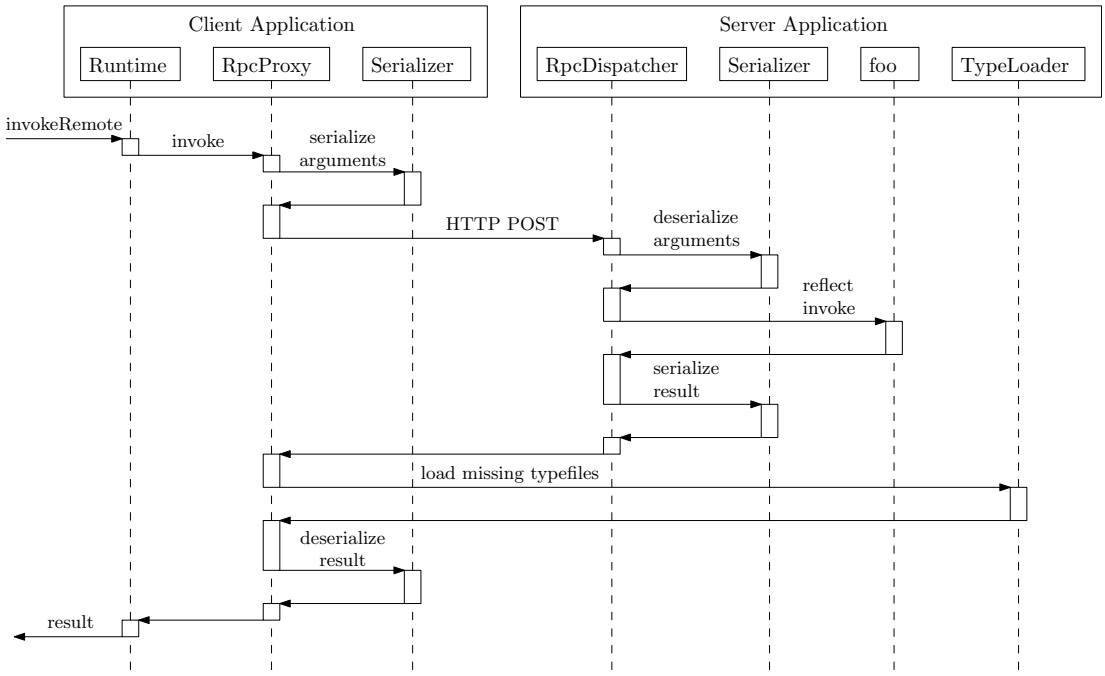


Figure 4.4: Remote method invocation.

1. The `invokeRemote` method is invoked.
2. Runtime delegates the call to the `swat.client.rpc.RpcProxy` class.
3. The method arguments are serialized as a tuple utilizing the client side serializer (`swat.client.json.JsonSerializer`).
4. A HTTP POST request is created and sent to the server.
5. On the server, the request is forwarded (the forwarding process is described in the following section) to the `swat.common.rpc.RpcDispatcher`.

6. Arguments of the remote method invocation are deserialized using the `swat.common.json.JsonSerializer`.
7. The remote method is invoked using reflection.
8. The return value is serialized and returned to the client as a string.
9. On the client, the `swat.client.rpc.RpcProxy` goes through the result and finds all types that are present in the result.
10. The `swat.common.TypeLoader` is asked for typefiles of types that are present in the result but not declared on the client yet.
11. The source code of the missing types is declared so there is nothing preventing deserialization of the result.
12. The result is deserialized and returned.

One more thing to notice is that the communication between the client `RpcProxy` and server `TypeLoader` is realized as a remote method invocation too. A return type of the `TypeLoader.get` is however always present on the client so it does not lead to a cycle.

4.6 Integration with a Web Framework

In order for the client application to be fully functional, the server side has to respond to a few requests that are generated by the Swat infrastructure on the client side. The toolkit was designed with a focus on a simple integration to existing web application frameworks so it is just necessary to forward the following three HTTP requests to the Swat server runtime.

- `GET /swat/tpe/<typeIdentifier>`
  Returns JavaScript code of the specified type.
- `GET /swat/app/<typeIdentifier>/<args?>`
  Returns JavaScript code of the specified type that mixes-in the `scala.App`. When executed in a browser the specified application starts running. The arguments are optional and allow simple parametrization of the application.
- `POST /swat/rpc/<methodIdentifier>`
  Returns result of the specified remote method invocation.

An examplary integration to the Play MVC Framework [25] can be seen in Listing 4.5. You may also notice that the toolkit uses Scala Futures [41] so when an operation blocks on e.g. system operation, the thread may be used to handle another request. As a consequence scalability of the whole application increases.

```
import scala.concurrent._
import ExecutionContext.Implicits.global
import play.api.mvc._
import swat.common._
import swat.common.rpc._

object Swat extends Controller {
    def tpe(typeIdentifier: String) = AsyncAction { r =>
        TypeLoader.getOrAlert(Array(typeIdentifier), Array.empty)
    }

    def app(typeIdentifier: String, args: String) = AsyncAction { r =>
        TypeLoader.getAppOrAlert(typeIdentifier, args.split(","))
    }

    def rpc(methodIdentifier: String) = AsyncAction { r =>
        val arguments = r.body.asJson.map(_.toString()).getOrElse("")
        RpcDispatcher.invoke(methodIdentifier, arguments)
    }

    private def AsyncAction(a: Request[AnyContent] => Future[String]) = Action { r =>
        Async(a(r).map(Ok(_)))
    }
}
```

Listing 4.5: Integration of Swat toolkit to Play Framework.

4.7 Build Process

The integration with SBT was initially planned as a plugin to SBT. A user of the toolkit would specify that his or her solution requires the "Swat SBT plugin", which would allow him or her to define not only standard Scala projects but also "Swat projects". SBT would download the plugin from the internet during the startup and the plugin would be responsible for invocation of the Swat compiler during compilation of a "Swat project".

The proposed setup would be nice, however it turned out that SBT plugins have to be binary compatible with SBT itself (i.e. use same version of Scala as SBT). The proposed "Swat SBT plugin" would have to use Scala 2.10.1 because it invokes the Swat compiler which uses Scala 2.10.1 itself. On the other hand the latest version of SBT (at time of writing the thesis) uses Scala 2.9 and therefore the plugin would be incompatible with SBT.

It has been solved by a new SBT action: besides the standard compilation action `compile` a new action `swat` is introduced. The `swat` action invokes the Swat compiler explicitly, but it is not that user-friendly and the compilation to JavaScript is not done in one step together with the standard compilation. As soon as SBT comes out with Scala 2.10.1 build, the proposed plugin should be implemented and the temporary hack replaced.

4.8 Tests

The most of the libraries in the toolkit are covered with tests with a purpose to early detect bugs that can be introduced to the compiler later. Especially the

`compiler` tests are useful, because a simple modification that seems correct may actually break the compilation of an AST when it is used in a different context. To run the tests start the SBT and execute the `test` action (i.e. write "test" and confirm it with enter).

5. Conclusion

With regards to the planned goals it can be stated the requirements on the toolkit have been met. The compiler is able to compile Scala programs to JavaScript and has no problems with any kinds of Scala language constructs. The interoperability with JavaScript is covered by quite a rich set of adapters and the possibility to use the native JavaScript code blocks inside the Scala programs. The runtime environment manages to support the compiled code during the execution while being consise. The toolkit was used to compile all the core Java and Scala classes necessary for its operation, moreover the futures and promises are fully supported. On the other hand, the supported library classes were added in a lazy-manner so there are only the core classes and other required classes. Mainly the lack of the Scala collection library (due to its size and complexity) is currently the most limiting factor. The integration to a web application framework is straightforward and the distribution of the libraries is as simple as it could possibly be.

The enhancement goals were also fulfilled. The JSON (de)serializer is usable both on the client-side and on the server-side. On top of the standard functionality it also allows to serialize object graphs containing reference cycles. The type loader is fully implemetned, RPC infractructure is conceptually solved and usable for a remote method invocation, but the support for more advanced scenarios involving authorization is not implemented yet.

To sum up, all the "big" challenges have been solved, some of them even to a near-production or production extent. On the other hand, the repetitive tasks that are not that interesting from the theoretical point of view, were given a lower priority, so the adapters and Scala library support have to be worked on.

5.1 Similar Tools

When the project started there have already been a few tools with the same or similar goal - to compile Scala to JavaScript. Namely ScalaGWT [42], s2js [44], Scalosure [45] and js-scala [46]. But it soon turned out that for a real web application they are not very usable. Members of the Scala academic community were also aware of that fact, therefore a new project Scala.js [47] was started during the work on the thesis.

ScalaGWT

The ScalaGWT is a new backend for the Scala compiler. The interesting thing is that in the first step it transforms Scala ASTs into Jribble [43], language, which is a new language described as a "looser" Java. The second step is to transform Jribble into Java. Finally, the resulting Java program is processed by the Google Web Toolkit [19], which transforms it into JavaScript. A motivation for the middle step (i.e. Jribble) is to open possibilities for other languages to

be transformed into JavaScript. And it is simpler to have Jribble as the target language instead of Java.

The drawbacks are that the project is abandoned for more than two years, is not well-integrated with Scala tools and frameworks and is rather in a prototype phase (the Scala library is not well supported). The author works primarily on the standard Scala compiler so ScalaGWT is a side-project for him.

S2js and Scalosure

Scalosure started as a fork of s2js, so these two tools are very similar to each other. Both of them are however unfinished, do not even support compilation of all Scala language constructs, the Scala library is mostly unsupported and the projects are not maintained. On the other hand they served as a great example on how to implement such a tool.

js-scala

The approach js-scala uses is a bit different but it is mentioned here for the sake of the completeness. It is a domain specific language that can be embedded into Scala programs. Its compiler plugin translates fragments of the DSL into JavaScript. The main disadvantage is that the existing code that should be transformed using js-scala has to be rewritten in order to follow rules of the DSL. The existing code cannot be used without modifications which is apparently the reason why Scala Library is unsupported. The project is currently rather a theoretical proof of concept than a tool focusing on a real world usage.

Scala.js

The project originated in January 2013 and is developed in EPFL (École Polytechnique Fédérale de Lausanne), which is a heart of the Scala community. Therefore it has big support from people that are directly involved in Scala language, compiler or Library. The compilation process works practically the same way as in Swat, the only difference is that the Scala.js compiler plugin runs after all the standard phases are executed.

The biggest achievement of this project is that the whole Scala Library was compiled to JavaScript and proven to be usable. The support of source maps is in progress and it has a tool for import of TypeScript adapter definitions to Scala. A downside is that the concept similar to Swat typefiles is not utilized so the compiled library has couple of megabytes after minification and compression.

5.1.1 Comparison

It is difficult to exactly quantify abilities of each of the aforementioned tools, becuase they are mostly undocumented. It is also difficult not to stay biased in

favor of the Swat toolkit but Table 5.1 tries to objectively summarize the level of maturity of each tool by different criteria:

Compiler — Scala language compiler support: ⋆ - major issues, ⋆⋆ almost everything supported, ⋆⋆⋆ - full support.

Interoperability — JavaScript interoperability: ⋆ - possible, ⋆⋆ adapters and native code, ⋆⋆⋆ - automatic coversion of adapters defined in different language.

Libraries — Standard library support: ⋆ nothing or basic classes, ⋆⋆ - not everything but important classes supported, ⋆⋆⋆ - whole library.

Distribution — Ease of library distribution and code reuse: ⋆ - not addressed (e.g. output to one file), ⋆⋆ at least somehow solved, ⋆⋆⋆ - typefiles and dynamic type loader.

Tooling — Build, debug, runtime and other tools: ⋆ - missing, ⋆⋆ - couple of tools, ⋆⋆⋆ - rich variety of tools.

Overall — Subjective overall impression.

Tool	Compiler	Interoperability	Libraries	Distribution	Tooling	Overall
ScalaGWT	⋆⋆	⋆	⋆	⋆⋆	⋆	⋆⋆
s2js	⋆	⋆	⋆	⋆	⋆	⋆
Scalosure	⋆	⋆	⋆	⋆⋆	⋆	⋆
js-scala	-	⋆⋆	⋆	⋆	⋆	⋆⋆
Scala.js	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆	⋆⋆	⋆⋆⋆
Swat	⋆⋆⋆	⋆⋆	⋆⋆	⋆⋆⋆	⋆⋆	⋆⋆⋆

Table 5.1: Comparison of Swat with other similar tools.

The fact that Scala.js and Swat have three stars in the Overall criterion does not necessarily mean they are ready for a production use. It means they are closest to it so they are already suitable for programmers to start experimenting with. At this moment there is no clear winner between Scala.js and Swat, both have their highlights and downsides so it is matter of who will sort them out faster. Scala.js currently has some support from the Scala community in EPFL so probably it has bigger chances of succeeding.

5.2 Future Work

As it is already apparent from the conclusion, one of the main future goals would be to continually add and extend support of Scala Library classes. Namely the collection library, which is for its richness widely used by Scala programmers. The adapters can be continually added in the same fashion as the Scala Library support. Or it is possible to reuse existing adapters defined in a different language (e.g. like Scala.js does with Typescript adapters).

The second more exciting direction is to explore ways how tasks that are not programmer-friendly in JavaScript could be simplified utilizing Scala. The grounds in form of the Swat toolkit have been laid so the really interesting applications can now be implemented or at least tried.

5.2.1 Web workers

JavaScript does not have any direct means of concurrent programming so it is quite challenging to implement an application that is computationally demanding and in the same time allowing the user to interact with it. A proof of this statement is that there are not many games or sophisticated visualization applications implemetned fully in JavaScript.

There is however a way how to execute code in parallel with the code that is being executed on a web page - by using WebWorkers [48]. The code in the standard execution context may create a new WebWorker which is sort of a different process. The standrd context and the worker context do not share access to memory so the only way of communication is to send or receive immutable serialized messages. The WebWorker library API is rather unfriendly so it is not widely used but applications that critacally need the concurrency have to bear with it, because there is no other option.

From a broader perspective WebWorkers are very similar concept to Actor Model [49] which is implemented in Scala within the Akka Concurrency Library [50]. There is an opportunity to create a class that would provide the Scala Actor API (or its essential subset) while under the hood it would use the unfriendly WebWorker API. A serialization and a deserialization of messages is already taken care of thanks to the JSON serializer.

Even a more interesting thing would be to abstract it a bit more so it would be possible on the client-side to create a "remote" actor, which would actually run on the server. The API of a "remote" actor would be the same as API of a "WebWorker" actor so for a programmer it would be undistinguishable whether the actor runs on a server or inside a WebWorker. It would also allow the web application to have some of its computation needs fulfilled by the server or the other way round.

There is another new feature in the JavaScript ecosystem called WebSockets [51], which would be necessary to utilize in case of the "remote" actors. A WebSocket allows to open bidirectional communication channel from the client side to the server side. They would also deserve to be given a better API that would hide low level details and make their usage friendlier.

5.2.2 Template Engine

Another thing that is currently rather inconvenient is templating in JavaScript. One often needs to create fragments of a HTML page programmatically, compose the fragments to bigger components and manipulate with them. The first option is

to create the whole fragment purely in JavaScript using DOM, but this approach is for larger parts of a page almost unusable. The second option is to use a template engine where it is possible to define a template with placeholders for data, compile the template to JavaScript, give the compiled template the data and make it render itself. This approach however brings the compilation step which does not pay off when used only for templating.

With Swat it should theoretically be possible to implement a type safe templating engine that would rely on Scala String Interpolation [52]. A template would be a function taking data and producing an HTML Element. A construction of the element and composition of the template from other templates would be a work of a new special string interpolator (e.g. `template`). A simplified example just to give an impression how it could look like is presented on Listing 5.1 where a model class (`User`) is defined together with a template for it.

```
case class User(name: String, frineds: List[User])

object Templates {
    def user(user: User): Element = template"""
        <div>
            <h2>${user.name}</h2>
            ...
            <h3>Friends</h3>
            ${user.friends.map(userTemplate)}
        </div>
    """
}
```

Listing 5.1: An example of templating engine.

5.2.3 Build Process

The Build process deserves the full integration to the SBT as it was not achievable at the moment of writing the thesis. An interesting improvement would be to employ the SBT resident compiler (a compiler that runs all the time, detects changes in the source files and recompiles the modified ones) so a programmer would not need to explicitly compile anything. He or she would just run the resident compiler and run the application. From that point it would be sufficient to modify a particular source file and reload the current page in order to immediately see the result.

Bibliography

[1] Introducing the Adobe Flash Platform. Adobe, July 2011.
http://www.adobe.com/devnet/flashplatform/articles/flashplatform_overview.html

[2] Java applet. Wikipedia, 2013.
http://en.wikipedia.org/wiki/Java_applet

[3] JavaScript. Wikipedia, 2013.
http://en.wikipedia.org/wiki/JavaScript

[4] ECMAScript Language Specification. Ecma International, June 2011.
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[5] Garrett, Jesse James.
Ajax: A New Approach to Web Applications. Adaptive Path, February 2005.
http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

[6] HTML5 Introduction. W3schools, 2013.
http://www.w3schools.com/html/html5_intro.asp

[7] Ashkenas, Jeremy. List of languages that compile to JS. July 2013.
https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS

[8] Meijer, Erik and Drayton, Peter. Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages. Microsoft Research 2004.
http://research.microsoft.com/en-us/um/people/emeijer/Papers/RDL04Meijer.pdf

[9] RequireJS.
http://requirejs.org/

[10] HeadJS.
http://headjs.com/

[11] JSLint, The JavaScript Code Quality Tool.
http://www.jslint.com/

[12] JSHint, a JavaScript Code Quality Tool.
http://www.jshint.com/

[13] Google Closure Compiler.
http://closure-compiler.appspot.com/home

[14] Dart: Structured web apps.
http://www.dartlang.org/

[15] CoffeeScript.
http://coffeescript.org/

[16] TypeScript.
www.typescriptlang.org/

[17] node.js
http://nodejs.org/

[18] Remote procedure call. Wikipedia 2013.
http://en.wikipedia.org/wiki/Remote_procedure_call

[19] GWT Project.
www.gwtproject.org/

[20] SharpKit - C# to JavaScript Compiler.
http://sharpkit.net/

[21] TIOBE Programming Community Index for July 2013. Tiobe, 2013.
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[22] Document Object Model (DOM) Specifications. W3C 2004.
http://www.w3.org/DOM/DOMTR

[23] jQuery.
http://jquery.com/

[24] Scala Build Tool.
http://www.scala-sbt.org/

[25] Play.
http://www.playframework.com/

[26] Typesafe.
http://typesafe.com/

[27] Ecma International.
http://www.ecma-international.org/

[28] Self.
http://selflanguage.org/

[29] Odersky, Martin and Zenger, Matthias. Scalable Component Abstractions. Proc. OOPSLA 2005.
http://lampwww.epfl.ch/~odersky/papers/ScalableComponent.pdf

[30] LINQ (Language-Integrated Query). MSDN 2013.
http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx

[31] Odersky, Martin and Spoon, Lex and Venners, Bill. Programming in Scala, Second Edition. Artima, 2010

[32] Advantages of Scala's Type System. StackOverflow 2013.
http://stackoverflow.com/questions/3112725/advantages-of-scalas-type-system

[33] McBeath, Jim. Scala Class Linearization. Coding and Life, 2010.
http://jim-mcbeath.blogspot.cz/2009/08/scala-class-linearization.html

[34] Symbols, Trees, and Types. Scala Documentation 2013.
http://docs.scala-lang.org/overviews/reflection/symbols-trees-types.html

[35] Seddon, Ryan. Introduction to JavaScript Source Maps. HTML5 Rocks, March 2012.
http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/

[36] World Wide Web Consortium.
http://www.w3.org/

[37] Scala Dynamic. Scala API Documentation 2013.
http://www.scala-lang.org/api/current/index.html#scala.Dynamic

[38] Scala Standard Library API. Scala API Documentation 2013.
http://www.scala-lang.org/api/current/index.html#package

[39] Lazy initialization. Wikipedia 2013.
http://en.wikipedia.org/wiki/Lazy_initialization

[40] Spoon, Lex. Writing Scala Compiler Plugins. Scala 2009.
http://www.scala-lang.org/node/140

[41] Futures and Promises. Scala Documentation 2013.
http://docs.scala-lang.org/overviews/core/futures.html

[42] Scala+GWT Project.
https://github.com/scalagwt

[43] Jribble. Scala+GWT Project.
http://scalagwt.github.io/jribble

[44] S2JS Compiler Plugin.
https://github.com/alvaroc1/s2js

[45] Scalosure.
https://github.com/efleming969/scalosure

[46] js.scala: JavaScript as an embedded DSL in Scala.
https://github.com/js-scala/js-scala

[47] Scala.js, a Scala to JavaScript compiler.
https://github.com/sjrd/scala-js

[48] Using web workers. Mozilla Developer Network 2013.
https://developer.mozilla.org/en-US/docs/Web/Guide/Performance/Using_web_workers

[49] Actor Model. Wikipedia 2013.
http://en.wikipedia.org/wiki/Actor_model

[50] Akka.
http://akka.io/

[51] Ubl, Malte and Kitamura Eiji. Introducing WebSockets: Bringing Sockets to the Web. HTML5 Rocks, Octomber 2010.
http://www.html5rocks.com/en/tutorials/websockets/basics/

[52] Suereth Josh. String Interpolation. Scala Documentation 2013.
http://docs.scala-lang.org/overviews/core/string-interpolation.html

[53] Odersky, Martin and Spoon, Lex and Venners, Bill. How to Write an Equality Method in Java. Aritma Developer, June 2009.
http://www.artima.com/lejava/articles/equality.html

Appendices

A. User Manual

The up to date version of Swat can be obtained from the Github Repository at `https://github.com/siroky/Swat`. You can either clone it into local directory using git or download it as a ZIP archive.

From the root directory go to the `swat` subdirectory and execute either `sbt.bat` (on Windows) or `sbt.sh` (on Unix). It should start SBT with the Swat project loaded in, but it may take couple of minutes because all dependencies of both SBT and Swat have to be downloaded in case they are not already present in the local Ivy cache.

When the "`[info] Set current project to swat`" appears, enter the `compile` command and confirm it with enter. It should compile the compiler and all other projects. Then execute the `swat` command in order to compile the runtime and the applications to JavaScript.

Finally when everything is compiled, it is time to run the server application. Switch to the `web` project using command `project web` and execute command `run`. Now you have couple of options. You can either access the sample application in your favorite web browser at `http://localhost:9000/` or access the playground application at `http://localhost:9000/playground` or visit the runtime test application at `http://localhost:9000/tests`.

Experimenting

If you want to experiment with the tool you may either directly use the sample appplication which serves that purpose or modify and extend the playground application. The best workflow is to start one SBT and run the `web` project there. Then start second SBT where you switch to the `playground` project (command `project playground`). You may modify sources of the `playground` any way you want, use the second SBT to compile them into JavaScript (command `swat`) and observe the outcome by refreshing `http://localhost:9000/playground`.

Tests

To run all tests, switch to the top level `swat` project (command `project swat`) and execute command `test`. You should immediately start seeing results of the tests.

B. Sample Application

The sample application that was developped using the Swat toolkit is called "Swatter". It allows the user to enter Scala code into one editor and compile it to JavaScript. The compiled code is displayed in an editor next to the Scala editor so the user can immediately see how some Scala constructs are compiled to JavaScript and what the output looks like. In case the compilation succeeds it is also possible to execute the compiled code. Screenshot of the sample application before compilation can be seen in the figure B.1.

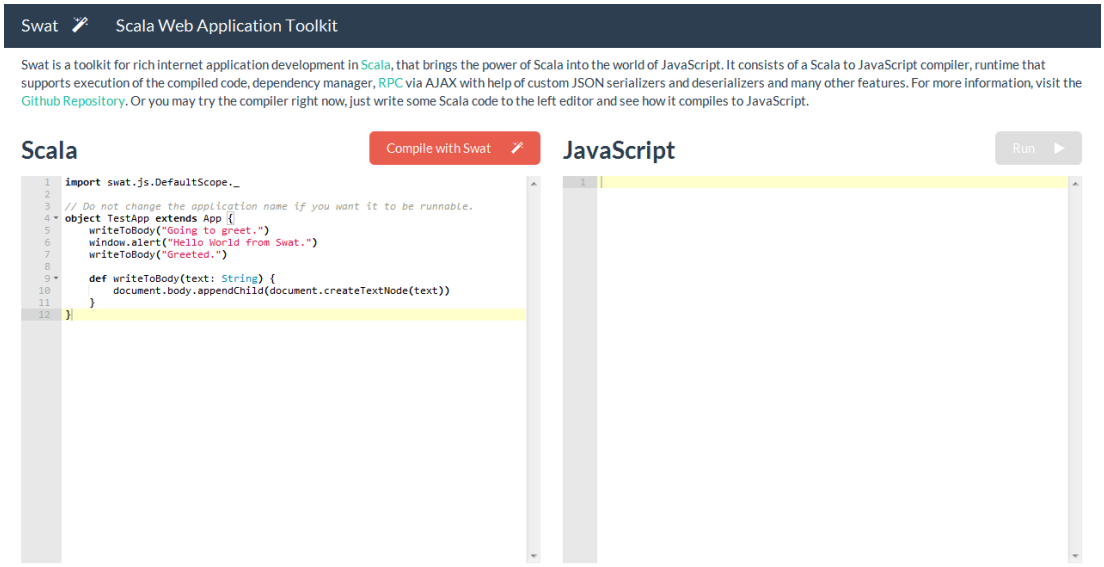


Figure B.1: Screenshot of the sample application.

When the "Compile with Swat" button is clicked, the application invokes the compiler on the server using RPC (figure B.2).

When the compilation finishes the result is shown in the JavaScript editor (figure B.3). It is also possible to run the compiled code using the "Run" button.

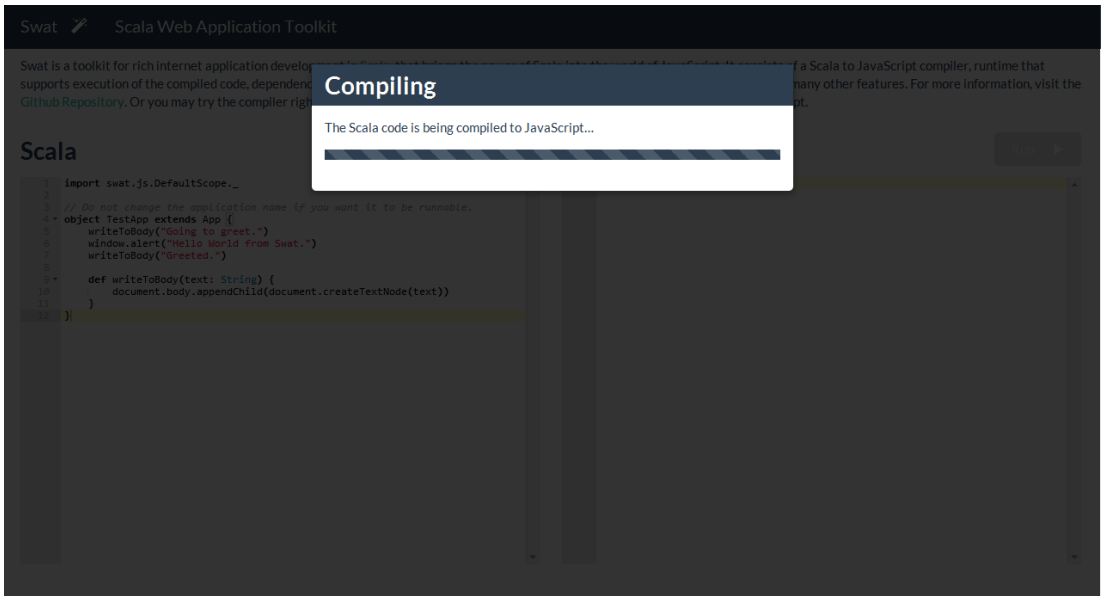


Figure B.2: Compilation to JavaScript in progress.

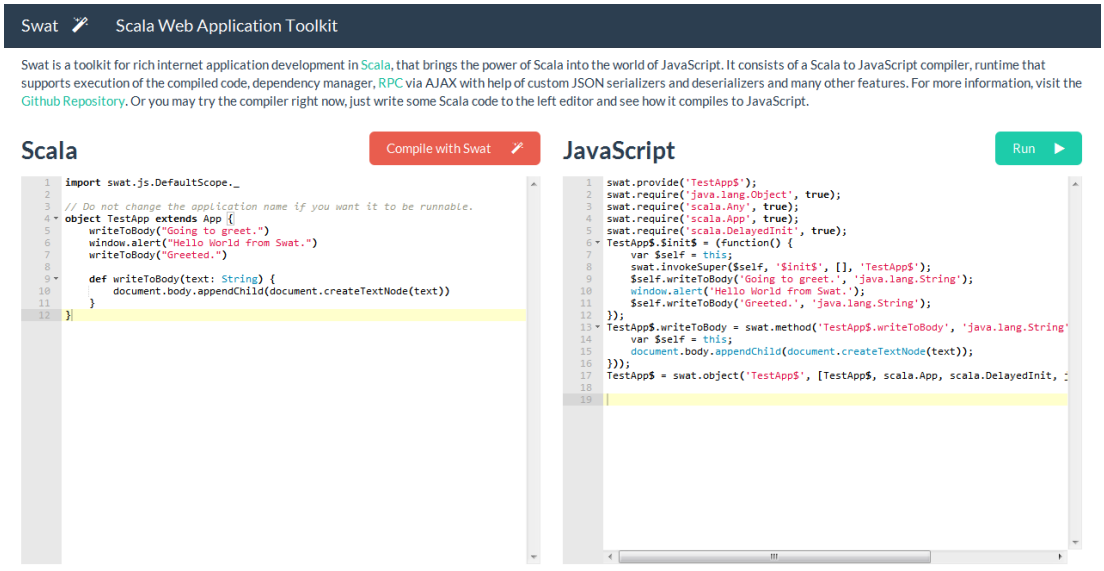


Figure B.3: Compilation result.

C. Content of attached CD

The attached CD contains the most current snapshot of the Github Repository available. There are two directories. The `thesis` contains the text of the thesis in PDF, the source of the thesis in TeX and all related resources. The `swat` contains the source code of the Swat toolkit, the SBT and generated project files for IntelliJ Idea.