# Extensible, fast and secure Scala expression evaluation engine

Roman Janusz

September 2014

**Abstract**

*Scripting and expression evaluation engines are popular tools in the Java software ecosystem. They're often used for purposes like defining document templates, enhancing various static configuration formats with dynamically evaluated snippets or defining data binding for user interfaces. However, most of these solutions employ dynamically typed languages and suffer from limited performance and lack of any concern for security. This effectively makes it impossible to use expression language as a feature exposed to end users. This paper presents a new approach to implementing an expression engine. It uses Scala as the expression language and leverages its static type system as well as its rich feature set to create an expression evaluation engine with expressive and concise language, high evaluation performance and fine grained security control.*

## I. Introduction

Embedding scripting environments inside larger software systems is a common technique used where the system serves as a container for some specialized, domain specific logic that can be loaded and executed dynamically. Such environments allow implementing features that cannot be expressed by simple graphical interfaces without losing much of fine-grained control and genericity provided by a scripting language.

Perhaps the most ubiquitous example of such a system is any web browser. The scripting language that it embeds is of course JavaScript and the browser servers as an execution environment for scripts.

Embedded scripting enables dynamic loading and execution of entire programs. A more lightweight and limited version of scripting engine is an expression evaluation engine. Instead of scripts that can be long and complex, an expression engine allows only simple expressions to be evaluated, with no access to constructs like loops, definitions of functions or classes, etc. Expressions usually consist of method or function invocations and operator applications that operate on expression input and produce some output data. Therefore, unlike script execution, expression evaluation is usually not allowed to cause any side effects.

One of the most challenging aspects of embedded scripting and expression evaluation engines is security. Loading and executing code provided dynamically by some third party poses a serious threat of letting malicious program execute which could result in either damage to the host system or unauthorized access to sensitive data. The more freedom a scripting engine allows, the harder it is to secure it. An expression engine is much easier to secure than a full-blown embedded scripting engine like browser-embedded JavaScript.

Good example of expression evaluation use case is an expression-enhanced administrative panel. Such panel may be used, for example, to adjust a greeting message that is displayed to users upon logging into web portal. The message would be configured as an expression (template) that has access to the data of the user logging in.

Another example could be an administrative interface for massive data processing system where an user could dynamically specify a data crunching job to be executed on a large amount of potentially distributed data (e.g. a map-reduce job). An expression would define actual transformations performed on that data.

This article describes an attempt to create an expression engine running on the JVM that is good for use cases like the ones described above. More systematically, the essential requirements for such engine are:

- very high evaluation performance - a single expression, compiled once, is likely to be evaluated massively, for many inputs
- fine grained security control - ability to easily reject potentially dangerous expressions by limiting what constructs and what API can be used
- flexibility and extensibility of the expression language

All of these requirements have a common denominator: to allow the expressions to be dynamically provided in runtime by means of graphical (e.g. administrative) interfaces without fear of security breaches and performance bottlenecks.

## II. Available JVM solutions

Scripting and expression evaluation engines are already very popular on the JVM. Some of them have been even standardized.

### I. Dedicated expression engines

JVM has a significant number of expression evaluation engine implementations. In this article, particular three are of our interest:

- Java Unified Expression Language (JUEL) [1]
- Spring Expression Language (SpEL) [2, ch.8]
- MVFLEX Expression Language (MVEL) [3]

JUEL is an implementation of the *Unified Expression Language* standardized as a part of JSR-245 specification - *Java Server Pages* [4]. In JSP and JSF (*Java Server Faces* [6], its successor) expression language is used mostly as a data binding layer between web interface and underlying business layer. Expressions are embedded inside JSP/JSF pages to specify what data various UI components link to. This is a good example of template-based user interface development, where a web page is a template that has some "holes" that need to be filled up in concrete context by expression evaluation.

SpEL is an expression language used by the *Spring Framework* [2], primarily for enhancing configuration of application components (called *beans*) in the IoC (*Inversion of Control*) container. In particular, expressions can be embedded in XML files that configure beans or inside Java annotations in source code of application components. This allows setting dynamic values to various configuration properties that can't be expressed by simple literal values.

MVEL is a general purpose expression engine with an hybrid dynamically-statically typed language and optional compilation to JVM bytecode which significantly improves performance.

The three listed expression engines use mostly dynamically typed, ad-hoc, simple expression languages. JUEL and SpEL were designed to be used in static resources of applications that use it. None of the three engines has any security features or means of easily achieving it.

## II. Expression engine on top of existing language

Perhaps the most well known specification of embedded scripting engine on the JVM is the JSR-233 standard, *Scripting for the Java Platform* [5]. It specifies standard data types and interfaces that must be provided by an engine that allows dynamic script execution inside a running JVM program. This standardized API primarily defines the way that host program can communicate with an embedded script - a glue layer. It is agnostic of the actual scripting language.

There is a vast amount of JSR-233 implementations for many standalone programming languages as well as custom, ad-hoc scripting languages. These include Java, JavaScript, Groovy, Scala, Ruby, Python, PHP, etc.

An JSR-233 scripting engine could be used as a basis for an expression engine providing that there is a way to limit the scripting language to simple expressions with proper security enforcement. The standard itself does not define any means to achieve that, but concrete implementations and the scripting language that they use may have such capability.

If a programming language has a standalone interpreter or compiler to JVM bytecode that by itself also runs under JVM, it could be used directly instead of relying on a JSR-233 implementation.

In general, suitability of an JSR-233 implementation or a standalone compiler for implementing an expression evaluation engine depends heavily on traits of the language itself. Since the new expression engine targets the JVM, we are only considering languages that have a JVM implementation. Also, requirements for evaluation performance and ability to statically analyze expression code suggest that languages with at least some form of static typing should more easily meet these requirements than dynamically typed languages.

Three languages have been chosen for evaluation in this article:

- Java [7] - statically typed but verbose and inflexible in many aspects
- Groovy [9] - concise and flexible in many ways, but mostly dynamically typed (static typing as an option)
- Scala [8] - statically typed, with conciseness and flexibility potentially on par with dynamically typed languages thanks to type inference and other features

## III. Language comparison

In order to choose a programming language that could serve as a basis of an expression evaluation engine, we must recognize what features in detail should the expression language provide. The

new expression engine should be considered as a potential replacement or alternative to already existing engines and therefore should support most of expression language constructs that these engines provide.

More in-depth research on engines mentioned earlier (JUEL, MvEL, SpEL) as well as a closer look on general purpose programming languages that could be a basis for the new expression engine reveals the following set of commonly supported constructs:

- essential constructs: literals, constants, arithmetic and logical operators
- variable assignments and references
- function and method calls
- class instantiation
- operators for concise access to array, list and map elements (like square brackets for array access in Java)
- concise syntax for array, list and map creation
- field references and assignments
- anonymous functions (lambdas)
- basic higher-order functions for collections like filtering and transformation
- automatic type coercion
- `null`-safe dereference, e.g. the `.?` operator in Groovy [10].

  Example: expression `obj.?field` will evaluate to `null` when `obj` is `null` instead of failing with an exception.
- default value operator (e.g. Groovy *Elvis* - `?:` [11])

  Makes it possible to concisely provide a fallback value that should be used when another value is `null`. Example: `obj.name ?: "unknown"`
- conditional expressions (e.g. ternary operator `?:` in Java)
- structural and imperative constructs (loops, `if-else` etc.)
- template expressions - string literals with "holes" that will be filled by actual expressions.

  An example of template expression in SpEL (see [2, p.189]): `My name is $person.name` where `person` is the input object containing a property `name`.
- standard C/C++ and Java like syntax for function and method invocations and operators (this excludes Lisp or Haskell like languages)

The features listed above are only the purely syntactical constructs used by the end user who will actually write expressions. However, there is also a set of additional features that are important for the programmer who sets up the expression evaluation context. In particular, he must be able to influence how expressions in concrete context are compiled and evaluated and what constructs, API and types are available to the author of expressions. In detail, these features include:

- ability to define what top-level functions and objects are available in the expression
- pluggable strategies for implementing dynamic object property access and method calls
- enhancing API of existing types, e.g. by adding additional methods to standard data types like strings
- limiting the API of well-known types only to a desired set of methods, fields, etc.
- defining additional automatic type conversions
- custom operators and operator overloading
- ability to statically (during expression compilation) analyze the expression code and reject potentially unsafe or forbidden constructs and invocations

JUEL, MVEL and SpEL engines as well as the languages Java, Groovy and Scala were all verified in detail if features mentioned above are supported by them in any form. Table **??** summarizes the results of this research.

As we can see, Scala and Groovy seem to be the most promising. They have most of the language features required for an expression engine and they are the only two technologies with possibility of static expression code analysis, which is probably the most important requirement for the new expression engine considering its need for security enforcement.

Additionally, after more detailed analysis, Scala seems to be the winning option over Groovy since the latter is severely limited in its static analysis capabilities by dynamic typing. Although it is possible to force static typing in Groovy and obtain some type information during compilation (by means of Groovy's `ASTTransformation` feature), this cripples other language features like dynamic method calls or ability to enhance API of existing types.

Therefore, Scala is selected as the most suitable technology for implementing the extensible, fast and secure expression engine described in this article.

## IV. Scala as expression language

In this section we will present a quick overview of essential properties of the Scala programming language and discuss in detail how its various features are sufficient to meet requirements discussed earlier. See [12] and [13] for more complete overview of the language.

Scala [8] is a programming language for the JVM aiming to blend the object oriented and functional programming paradigms and retain good interoperability with Java. It is a statically typed language with type inference and very complex, fine-grained type system. At the same time it aims to be close in its conciseness and flexibility to dynamically typed programming languages.

| **Feature** | Java | Groovy | Scala | JUEL | SpEL | MVEL |
|---|---|---|---|---|---|---|
| constants and literals | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| arithmetic and logical expressions | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| variables | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| object property access and assignment | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| method calls | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| class instantiation | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ |
| concise collection and map element access | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| concise collection and map creation | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| basic higher order functions for collections | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| lambda expressions | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ |
| automatic type coercion | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| `null`-safe dereference | ✗ | ✓ | ✓ | ✗ | ✓ | ✓ |
| default value operator | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ |
| conditional expressions | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| imperative and structural statements | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ |
| template expressions | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| pluggable dynamic property access and assignment strategies | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| pluggable dynamic method invocation strategies | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ |
| enhancing API of existing types | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ |
| limiting API of existing types | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ |
| custom automatic type conversions | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ |
| custom operators and operator overloading | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ |
| static expression code analysis | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ |

**Table 1:** *Language comparison summary*

Scala version 2.10 introduces important metaprogramming facilities - reflection and macros which are key for implementing security features of the expression engine.

Scala naturally supports essential constructs like constants, arithmetic expressions, object property access and method calls, constructors, etc. Expressions in Scala mostly have a syntax similar to Java. Differences include e.g. usage of square brackets instead of angle brackets to denote generic types. For example, `Arrays.<String>asList("str")` in Java becomes `Arrays.asList[String]("str")` in Scala (the type can however be usually omitted due to type inference).

In Scala, there is no clear difference between a method and an operator. Every operator can be though as if it's a method with a symbolic name. At the same time every method with takes exactly one argument can be called with an infix syntax. All of the following constructs are correct:

```
1 + 2
1.+(2)
set.contains(element)
set contains element
```

An unique feature of Scala is an ability to define *implicit conversions* [12, ch.15] between types. In an expression evaluation engine, this can be used for two purposes:

- to provide custom automatic type coercion
- to enhance API of existing types with new methods, fields, etc.

The API enhancing is possible thanks to the so called *implicit views*. By introducing an implicit conversion from type `A` to type `B` we effectively enhance API of `A` with API of `B`. Methods in type `B` become *extension methods* of type `A`. This is often realized in Scala using a syntactic sugar called `implicit classes`. The following example enhances type `String` with an additional `capitalize` method:

```
implicit class richStr(str: String) {
  def capitalize =
    str(0).toUpper + str.substring(1)
}
```

It is worth noting that extension method can also be called with an infix syntax, so it is effectively possible to create and overload operators.

Scala also has a number of syntactic sugars that increase conciseness. For example, when a method is called `apply`, it can serve as an overloaded function application operator, i.e. `obj.apply(arg)` can be rewritten just as `obj(arg)`. Thanks to this, we can have concise collection creation constructs like:

```
List(1, 2, 3, 4)
Array("str", null, "sth")
Map("one" -> 1, "two" -> 2)
```

The example above also uses a few other, independent Scala features:

- variadic arguments (the `apply` method takes a variable number of arguments of the same type)
- first-class objects (`List`, `Array` and `Map` are actually singletons associated with their corresponding collection types - so called *companion objects*)
- type inference (the `apply` methods on objects `List`, `Array` and `Map` are generic)
- custom operators (the `->` is a custom operator that creates a pair out of its operands)

The `apply` method can also be used to implement concise access to collection elements, e.g. `someList(2)`, `someMap("key")`.

It is important to remember that implicit conversions are resolved statically, based on types known at compile-time (unlike type coersions in JUEL/SpEL/MVEL which use runtime type information).

Scala is a functional language, so it naturally supports constructs like lambda expressions and higher order functions. Examples:

```
List(1, 2, 3, 4).filter(_ % 2 == 0)
List(1, 2, 3, 4).fitler(i => i %2 == 0)
List(1, 2, 3, 4).map(_.toString)
```

Scala does not have a ternary conditional operator (`?:`) from Java but it does not explicitly need it, because - unlike in Java - the `if-else` statement is an *expression* in Scala which means that is has a *value*. This means that `if-else` can be used instead of ternary operator:

```
val yes: Boolean = ???
val repr = if(yes) "yes" else "no"
```

Scala also does not have `null`-safe dereference operator or default value operator, but it is possible to create a custom operator in Scala that serves both purposes. We will not show the implementation in this article, but as a guideline, the technique used to do this involves either *macros* or an implicit conversion that takes its input as an *by name argument* (another distinct feature of Scala). As a result, following syntax is possible:

```
obj.prop.method(arg) ? defaultValue
```

The expression in above example will evaluate to `defaultValue` either when left-hand-side operator evaluates to `null` or when evaluation of left-hand-side would result in a `NullPointerException`. So the `?` operator is actually a combination of `null`-safe dereference and fallback value operator.

In version 2.10, Scala introduced *string interpolation* syntax. It allows to splice identifiers and expressions into a string literal instead of concatenating all the parts with the + operator. The syntax happens to be similar or identical to template expression syntax in JUEL/SpEL/MVEL and therefore, string interpolations can directly serve as a realization of template expressions.

```
val name = "John"
val surname = "Kowalski"
val greeting = s"Hello, $name $surname!"
```

An area where Scala is perhaps more limited than dynamically typed expression languages is the ability to provide pluggable strategies for dynamic property access and method invocations on arbitrary types. The limitation is caused primarily by static typing in Scala. However, this feature is often used in JUEL/SpEL/MVEL as a method for API enhancement which has already been shown to be possible in Scala using implicit classes.

Scala allows dynamic method and property access on types that the programmer has control of (is their author). There is a special *trait* (base type) in Scala - `scala.Dynamic` that can be extended by classes that need to support dynamic property/method access. In order to actually provide the property or method resolution strategy, that class needs to implement `selectDynamic` and `applyDynamic` methods.

It is important to remember that `selectDynamic` and `applyDynamic` still are statically typed - they have a statically typed arguments and a static return type. So this feature cannot be used to somehow introduce dynamic typing into Scala.

## V. Metaprogramming in Scala

Metaprogramming is the method used by Scala-based expression engine to enforce security of evaluated expressions. Therefore it deserves a separate overview in this article.

### I. Scala compiler

Metaprogramming techniques in Scala are heavily related to internal architecture of the Scala compiler. Therefore we will briefly describe the compilation process and data types used by the compiler to represent various entities in compiled programs. These data types are also exposed by the reflection API, which is used in both runtime reflection and macros (described later).

A compiler run consists of several *phases* [14]. Each phase takes as a input the output of previous phase. In most cases, that input and output is the *abstract syntax tree* (AST) of the program. Each phase performs some modifications and refinements on the AST. In total, there is about 30 phases. Phases can be divided logically into two stages, according to standard, generic architecture of compilers:

- Stage of *analysis* is responsible for extracting as much information from the textual source code and transform it into canonical intermediate representation ready to be compiled into JVM bytecode. The most important phases of analysis are the `parser`, `namer` and `typer`. Parser transforms the textual code into an initial AST. Namer is responsible at least for resolution of imports and typer - the most complex and time consuming phase - performs the typechecking of the code.
- Stage of *synthesis* gradually transforms the AST into more raw forms, closer to the final representation which is ultimately translated into JVM bytecode. It contains several phases which are not of interest in this article.

The compiled program is represented inside the compiler primarily by three types of objects which refer to each other:

- `Tree` is a representation of *abstract syntax tree* that is an input and output of most of the phases. Trees have immutable, hierarchical structure and mutable attributes. The structure

of the tree heavily depends on what phase is being executed. For example, a tree just after parsing is closest in its structure to textual source code - no syntactic sugars have been expanded, no types inferred, no implicit conversions applied, etc. On the other hand, all of this information is explicitly represented in the tree after it exists the typechecker phase.

- `Type` is a representation of every Scala data type. Types are one of the attributes associated with most of the trees after they are typechecked. Types have a rich API which, among other things, makes it possible to inspect what members are available on various types or perform operations like type conformance tests.
- `Symbol` objects represent various entities found in the source code. For example, each variable, method, class, object, etc. has a symbol. Symbols are divided into two subcategories - term symbols (values, variables, objects, etc.) and type symbols (classes, traits, abstract types, type aliases, etc.). Symbols are also associated with most of the trees after typechecking. For example, a tree representing a method invocation will have the symbol of the method being invoked associated with it.

## II. Reflection and macros

Trees, types and symbols are not only used by the compiler internally, but they are also exposed by the Scala reflection API [15]. This API is used by two metaprogramming facilities in Scala - runtime reflection an macros.

*Runtime reflection* [17] allows the programmer to work with trees, types and symbols in runtime, as the name suggests. It is possible to obtain types and symbols for runtime objects and extract various information from it.

*Macros* [16] is a form of compile time metaprogramming in Scala. A macro is a special declaration in the Scala source code which is syntactially similar to a regular method. The difference between a macro and a method is that while method works in runtime - on values of its arguments and returns some other value - a macro is invoked during compilation and takes as its input and output a `Tree` for each argument. It can inspect these trees and modify them before returning a final tree that will replace the macro invocation and eventually be actually compiled to JVM bytecode. Macros are expanded inside the typechecker phase. This means that the trees that macros work on carry maximum information, including types and symbols. Macros can also trigger compilation errors during their expansion.

Macros are the key feature for implementing security in a Scala based expression evaluation engine. They effectively allow static analysis of expression code which can inspect what language constructs are being used in expression being compiled and what methods are being invoked by it. Upon detection of a forbidden construct or invocation, the macro can simply trigger a compilation error.

# VI. Scala expression engine implementation

This section describes the core API of Scala expression evaluation engine as well as gives more details about the underlying implementation.

# I. Essential components and types

The central component of the Scala expression engine is the `ScexCompiler` object. It encapsulates the actual Scala compiler and provides the toplevel API for expression compilation. The most important method for expression compilation has following signature:

```
import scala.reflect
  .runtime.universe.TypeTag

def getCompiledExpression[
  C <: ExpressionContext[_, _] : TypeTag,
  T: TypeTag](
    profile: ExpressionProfile,
    expression: String,
    template: Boolean = true,
    header: String = ""): Expression[C, T]
```

The types used in above signature and other API elements are:

- `TypeTag` is a type from Scala runtime refletion API. The syntax `T: TypeTag` denotes that method `getCompiledExpression` will have access to runtime information about type `T`.
- `Expression[C,T]` is a function that takes an expression context (type `C`) as an input and returns some some arbitrary value of type `T`. This is what we understand as *expression evaluation*.
- `ExpressionContext[R,V]` is the input of the expression and serves two purposes:
  - It encapsulates the *root object* of the expression - an object of arbitrary type `R` whose API is directly exposed to be used by expression code. Root object is also *de facto* the actual input data of the expression.
  - It serves as a container for *variables* of arbitrary type `V` which can be accessed inside the expression.
- `ExpressionProfile` customizes the way expression is compiled. The profile consists of:
  - `SyntaxValidator` defines what language constructs are allowed to be used by the expression.
  - `SymbolValidator` defines which methods can be invoked and which fields can be accessed for particular types.
  - Expression header - code that will be compiled along each expression. Header is primarily used to provide `import` clauses for expression code.
  - Expression utils - implementation of additional API that can be used inside expressions. This can be used to implement some API in Scala in situation when application using the expression evaluation engine is by itself written in Java.

  Symbol validator and syntax validator are the core components regarding static analysis of expressions and security enforcement. The way they are created and configured will be described in more detail later in this article.

- The `template` parameter controls whether an expression is compiled as template expression.
- The `header` parameter provides additional header that will be compiled in expression code. Its purpose is similar to the header specified in expression profile, but can be different for each expression that uses the same profile.

## II. Compilation process

Expression compilation is performed in following steps:

1. `ExpressionDef` object is created. This intermediate object contains full information about how the expression should be compiled, what the expression code is and what are the input and output types.
2. `ExpressionDef` object is optionally preprocessed. Preprocessing may include any modification of expression definition (e.g. additional pre-translation of code)
3. Expression code is wrapped into a proper Scala source file that contains the expression class implementing `Expression` trait. The bare expression code is wrapped into an invocation of security-enforcing macro.
4. Source file is passed into the Scala compiler for actual compilation.
5. During `typer` phase, the security-enforcing macro is expanded. The macro uses syntax validator and symbol validator to analyze expression code for forbidden constructs and invocations.
6. Expression class is compiled into JVM bytecode and `.class` files are generated.
7. Expression class is loaded into the JVM, instantiated and returned as a compiled expression.

## III. Configuration of validators

As described earlier, the programmer can decide what language constructs and invocations can be used inside expression code by providing a `SyntaxValidator` and a `SymbolValidator` instance through `ExpressionProfile`. This section describes how these two components are configured.

Syntax validator is a following *trait*:

```
import scala.reflect.macros.Universe

trait SyntaxValidator {
  def validateSyntax
      (u: Universe)(tree: u.Tree):
      (Boolean, List[u.Tree])
}
```

The `tree` is a fully typechecked tree representing the expression code. Since it contains type information, it may be used for much deeper validation than just syntax validation. However, validation of types and symbols is much more complex in its implementation and was implemented

as a separate component, the symbol validator. Syntax validator is intended to be used to look for forbidden syntactic constructs, like definitions or loops. Such validation can be easily implemented using pattern matching on particular cases of trees. This is what is usually done inside the `validateSyntax` method.

The `validateSyntax` method returns a pair that denotes whether the tree passed the validation and a list of children trees that needs to be validated next. This way syntax validation descends through entire expression tree.

By default, expression engine provides a standard implementation of syntax validator that allows only simple expressions to be used. This includes constants, identificators, method calls, operator applications, lambda expressions, conditional statements, simple blocks and constructor invocations. No assignments, declarations, definitions or loops are allowed.

Symbol validator is much more complex to implement and requires some more convenient configuration layer to avoid dealing with raw `Tree` objects.

In order to create an instance of `SymbolValidator`, the programmer needs to provide an ACL-like structure (*Access Control List*). In a program using Scala expression engine, this ACL is represented by type `List[MemberAccessSpec]`. Each element, a `MemberAccessSpec` instance either allows or denies usage of some symbol (method, field or constructor, including methods visible through *implicit views*) on a particular type. In detail, the `MemberAccessSpec` contains following information:

- information about the type
- signature of the member of that type that is being denied or allowed by this element
- optional signature of implicit conversion that is used to obtain an *implicit view* that causes that member to be available on that type
- information about whether this element denies or allows usage of given symbol on given type

## IV. Symbol validation DSL

Instances of `MemberAccessSpec` and the ACL are not created by the programmer manually. Instead, a dedicated Scala DSL (*domain specific language*) has been created for that purpose. That DSL is an example of a technique called *language virtualization*, which essentially means to replace standard semantics of some language (in this context Scala) with some custom semantics, i.e. leverage syntax and type system of Scala for a different purpose than compilation of Scala program.

Language virtualization in symbol validation DSL is realized by another set of Scala macros, `allow` and `deny`. Each of these macros take a specific block of code (format described below) and generate code that evaluates to a part of the ACL. These parts can be later concatenated into full ACL using standard Scala list concatenation operator `++`.

The blocks being passed as inputs to `allow` and `deny` contain a references to scala types and their members that the programmer wants to allow or deny. References are represented as lambda expressions or simple method calls (i.e. lambda expressions and method calls are *virtualized* as references to symbols).

If a programmer wants to allow or deny usage of some static Java member or member of Scala toplevel object, the reference to that member is put directly into the block passed as input to `allow`

or `deny`. If the programmer wants to allow or deny usage of non-static member on some type, that reference must be additionally put inside an `on` statement which specifies the exact type on which the member is allowed or denied.

Simple example of ACL created with symbol validation DSL:

```
val acl = allow {
  String.valueOf(_: Int)
} ++ deny {
  on { obj: java.lang.Object =>
    obj.wait()
    obj.notify()
    obj.equals _
  }
}
```

In example above, the `acl` value will have the type `List[MemberAccessSpec]` which can be concatenated with other ACL parts or eventually used to create a symbol validator. The above example allows usage of static method `valueOf` that takes a single `Int` argument of class `java.lang.String` and denies instance methods `wait`, `notify` and `equals` on type `java.lang.Object`.

As in all ACL-like structures, if more than one element in the ACL matches invocation that is being validated, the element earlier in the list has priority over the latter one.

Symbol validator DSL also provides additional convenience constructs for allowing or denying entire sets of members with single statements. Below is presented a comprehensive list of all available constructs in the DSL (examples are assumed to be inside `allow` or `deny` block):

- static members (explicitly)

  ```
  Integer.parseInt(_: String)
  ```

- instance members (explicitly) - either available directly or by an implicit view

  ```
  on { str: String =>
    str.substring(_: Int)
    str.substring(_: Int, _: Int)
    str.toString()
  }
  ```

- all static members of some Java class

  ```
  allStatic[String].members
  ```

- all static members of some Java class with given name

  ```
  allStatic[String].membersNamed.valueOf
  ```

- all members available on given type

```
on { str: String =>
  str.all.members
}
```

This deliberately excludes members declared in toplevel types `Any` and `AnyRef`, i.e. `equals`, `hashCode`, `==` etc. since they are available on all types (or all reference types) and it's not usually the intention of programmer to include them.

- all members available on given type with given name

```
on { str: String =>
  str.all.membersNamed.substring
}
```

- all members available on given type declared in class representing that type

```
on { str: String =>
  str.all.declared.members
}
```

This includes all members declared in that class directly and all members that this class overrides.

- all members available on given type declared in class representing that type and not inherited from supertypes

```
on { str: String =>
  str.all.introduced.members
}
```

This includes all members declared in that class that don't override a member from superclass.

- all members available on given type by an implicit view to some type

```
on { str: String =>
  str.implicitlyAs[StringOps]
    .all.members
}
```

- a single constructor for given type

```
new java.util.Date
```

- all constructors for given type

```
on { str: String =>
  str.all.constructors
}
```

- all bean getters, bean setters, scala getters or scala setters available on given type

```
on { jb: SomeJavaBean =>
  jb.all.beanGetters
  jb.all.beanSetters
}
on { sc: ScalaClass =>
  sc.all.scalaGetters
  sc.all.scalaSetters
}
```

Constructs listed above can be freely combined to form more complex patterns, e.g.

```
on { o: SomeType =>
  o.implicitlyAs[OtherType]
    .all.introduced.beanGetters
}
```

## V. Detailed security analysis

Syntax and symbol validators provide means to achieve tight security enforcement, but they must be used carefully and with full awareness of possible vulnerabilities. Every security feature protects only against some particular class of attacks. This section describes possible security vulnerabilities that could be exposed by expression engine and makes it clear which of them the Scala expression engine protects against. It also discusses which security problems must be addressed separately, in other layers of the software that uses an expression engine.

Exposing expressions to a malicious user could generally cause three types of security breaches:

- unauthorized access to sensitive data
- unauthorized access to operations that could damage or compromise the host system
- exhaustion of resources used by the host system or abnormally high utilization of these resources

All of these breaches could obviously be caused by exposing an expression API that is insecure by itself. It is a responsibility of the programmer/administrator to ensure that functions and operations that can be used in an expression are internally secure. If they aren't, the engine itself has no way of knowing this, since it only uses compile-time information for validation. So the most basic rule of designing secure expression API is to make sure each exposed method is secure by itself.

However, sometimes it may not be obvious that some operation is insecure. For that reason, we give examples of common ways in which a member of expression API may become a security hole.

- Exposing `toString()`

  Allowing `toString()` calls is an easy way to leak sensitive information. The `toString()` method should only be allowed on simple data types that have an obvious implementation of

`toString()`. Therefore, one should be extremely careful when allowing calling `toString()` on:

  - classes encapsulating something else than immutable data, e.g. some internal state
  - classes and interfaces that are a base of an open type hierarchy - even if the base type itself is secure, any of its subtypes may no longer be so
  - generic containers, e.g. collections

  For example, allowing calling `toString()` on type `List[Any]` is an obvious security issue. `List`'s `toString()` implementation internally calls `toString()` on its elements and the expression engine doesn't know about it, because this information is not available in compile time. This effectively gives us access to results of `toString()` on any value that could be inside a list. `List[Any]#toString()` is therefore as insecure as `Any#toString()`.

- Exposing API that internally uses `toString()`

  In any API which exposes string manipulation methods, there usually are some functions and operators which take arbitrary values as their arguments and call `toString()` on them internally. The `+` operator for concatenation of strings and arbitrary values is perhaps the most apparent example of such API. Therefore, when exposing these API fragments, one must be as careful as with the `toString()`

- Exposing untyped operations

  It is highly recommended that the exposed expression API is as strongly, statically typed as possible. This way we have much more information to work with in compile time and we can be vastly more precise when defining security rules using symbol validator DSL.

- Exposing API with non-constant time or memory complexity.

  This could lead to easy exhaustion of host system resources, especially considering the fact that expressions are meant to be evaluated massively, on multiple pieces of input data.

  Even more dangerous than exposing methods with just non-constant (e.g. linear) complexity is exposing APIs with complexity determined by *runtime values* passed to them. For example, Scala standard library provides an `*` operator on the `String` class which takes an integer argument and replicates a string given number of times. Simply by passing a large integer value, e.g. `"abc"*10000000`, expression writer can easily cause allocation of very large amounts of memory.

Apart from securing the API itself, one must also take some measures to secure the syntax. Scala is a general-purpose programming language, but the Scala expression engine enables only a small portion of its syntactic constructs. This is done on purpose, to disallow elements which could easily exhaust resources of the host systems (e.g. loops). Although it is possible to provide custom syntax validator and allow more, it is highly recommended to stay with the predefined simple-expression syntax subset of Scala. In other words, we should keep the expression engine an *expression* engine and not make it a scripting engine, which is much harder to secure.

Finally, one must be aware that usage of CPU and memory during expression evaluation is at least proportional to the overall complexity of the expression. Therefore, some limits must be

enforced in this area. For example, one may want to limit the textual length of an expression or maximum depth of its syntax tree. The latter case can be easily implemented with a custom syntax validator.

## VII. Usage example

This section outlines a typical use case for Scala expression engine and explains how its features make it a viable solution that retains good performance and enforces tight security.

### I. Dynamic serving of configuration files

Scala expression engine could be used in a system for mass management and configuration of devices, e.g. telecommunication devices like routers. These devices usually require dynamic, remote reconfiguration. This may be easily done with configuration files periodically fetched by devices from the management system. However, contents of configuration files may need to be generated dynamically, based on some data associated with particular device asking for configuration. Additionally, the exact format of each config file may need to be different for various device types. Above all this, system operators may want to change the contents and structure of served configuration files on a regular basis, without reconfiguring and restarting the management system itself.

This problem can be easily addressed with an expression engine: operators could use some user interface to define *configuration file templates*. Each template may have placeholders that would be filled by some data associated with the device when that device asks for configuration file. In other words, each template would be an *expression* that evaluates to exact configuration file contents based on particular device data.

For example, let's assume that the data associated with a device is represented by following Scala trait:

```
trait DeviceData {
  def id: String
  def currentIP: String
  def modelName: String
  def manufacturer: String
  def wifiSSID: String
  def wifiPassword: String
  def wifiEncryption: String
}
```

This data is either sent by the device when it asks for configuration file (e.g. current IP) or is stored in the system database (e.g. WiFi parameters to be set). It may also come from some external systems.

Let's assume that we need to send configuration file to a group of home routers in order to setup the WiFi network. The configuration file format accepted by devices may look like this:

```
[wifi]
ssid=homeWifi
```

```
password=secretPassword
encryption=WPA2
```

System operator could then define a template for such configuration file. This template would be a SCEX expression that could look like below. This is assuming that the programmer has defined the expression API for this particular purpose to contain a value `deviceData` which is of type `DeviceData`.

```
[wifi]
ssid=${deviceData.wifiSSID}
password=${deviceData.wifiPassword}
encryption=${deviceData.wifiEncryption}
```

However, some types of devices may not accept the values in the same format as they are stored in the system database. For example, let's assume that encryption mode saved in database is exactly "WPA2", but the device accepts only lowercase "wpa2". System operator can quickly define a separate template for this particular type of device and adjust to device behavior:

```
encryption=${deviceData.wifiEncryption.toLowerCase}
```

Thanks to the fact that an expression engine allows us to dynamically define small snippets of code, the operator has very detailed control over what exactly is being sent to devices. Also, since such strange issues may be very unpredictable, it is very hard to preconfigure the management system to handle all device types properly. The possibility of dynamic loading and redefinition of templates is very useful here.

Use case described above has all the important needs that SCEX addresses:

- templates (expressions) are defined and loaded dynamically
- templates are defined by system users, possibly using graphical interfaces - security enforcement is needed
- single template is typically evaluated and applied on large number of devices - evaluation performance is important

## VIII. Performance

A small set of performance tests was carried out to confirm that the evaluation of expressions in the new engine is comparable to performance of raw bytecode. The engine was compared with statically compiled Scala code and engines described earlier - JUEL, SpEL and MVEL. Also, a simple compilation performance test was carried out.

The purpose of tests is not to give any precise numeric coefficients since evaluation and compilation performance greatly depends on what kind of expressions are actually compiled and evaluated.

The tests consisted of series of evaluations or compilations of expressions in a loop. Tables below present the results. The numbers are in milliseconds and serve only for comparison with each other.

Evaluation

| | |
|---|---|
| Scala | 1312 |
| Scala engine | 2312 |
| SpEL | 150565 |
| JUEL | 161567 |
| MVEL | 14627 |

Compilation

| | |
|---|---|
| Scala engine | 115868 |
| SpEL | 51 |
| JUEL | 56 |
| MVEL | 1039 |

The tests show that evaluation performance is indeed close to raw Scala code compiled statically, but the price is a heavyweight compilation process. This is however well-suited for requirements defined in this article - a single, compiled expression is expected to be evaluated massively, on multiple inputs.

## IX. Summary

We have presented a new approach to implementing an expression evaluation engine for the JVM. The key goals of that approach is high evaluation performance and secure compilation and evaluation of expressions so that they can be safely created by potentially malicious users, possibly using graphic system interfaces. These properties should not result in a less rich or flexible expression language than ones provided by existing solutions.

We have shown that already existing solutions, represented primarily by engines JUEL, SpEL and MVEL do not meet these requirements. We have also evaluated already existing, general purpose programming languages for suitability of building an expression engine on top of them. The language of choice was Scala. Its primary advantage over other languages is static type system which guarantees good performance and makes it possible to statically analyze code using macros, a compile metaprogramming facility. Despite being statically typed, Scala still retains much of the flexibility and conciseness of dynamically typed languages.

We have described how exactly Scala features can be leveraged for an expression language and presented a quick overview of its metaprogramming capabilities. Then we have outlined the most important elements of API and architecture of the new expression engine. As one of the most important elements, a DSL for specifying what invocations are allowed inside expression code was presented in more detail.

Finally we have shown that the new expression engine meets its performance requirements by being comparable to statically compiled Scala code.

## References


[1] *Java Unified Expression Language*. `http://juel.sourceforge.net/juel.pdf` access: September 2014

[2] R. Johnson et al. *Spring Framework Reference Documentation, 3.2.4.RELEASE*. 2013

[3] *MVFLEX Expression Language*. http://mvel.codehaus.org/
access: September 2014

[4] K.M. Chung *JavaServer Pages™ Specification Version 2.3, Maintenace Release 3*. May 2013

[5] M. Grogan *JSR-223 – Scripting for the Java Platform. Final Draft Specification, version 1.0*. Sun Microsystems, 2006

[6] E. Burns *JavaServer™ Faces Specification, Version 2.2*. Oracle America, Inc, March 2006

[7] J. Gosling et al. *The Java®Language Specification, Java SE 7 Edition* Oracle America, Inc, February 2013

[8] M. Odersky *Scala Language Specification, Version 2.8*. Programming Methods Laboratory, EPFL, Wrzesień 2013

[9] *The Groovy programming language* http://groovy.codehaus.org/
access: September 2014

[10] *Groovy - Safe Navigation Operator* http://groovy.codehaus.org/Operators
#Operators-SafeNavigationOperator
%28?.%29
access: September 2014

[11] *Groovy - Elvis Operator* http://groovy.codehaus.org/Operators
#Operators-ElvisOperator%28?:%29
access: September 2014

[12] M. Odersky *Scala by Example* Programming Methods Laboratory, EPFL, June 2014

[13] M. Odersky, L. Spoon, B. Venners *Programming in Scala: a comprehensive step-by-step guide*, 1st edition. Artima Press, 2008

[14] *Overview of Scala Compiler Phases*
https://wiki.scala-lang.org/display
/SIW/Overview+of+Compiler+Phases
access: September 2014

[15] *Scala Reflection API: Symbols, Trees and Types*
http://docs.scala-lang.org/overviews
/reflection/reflection/
symbols-trees-types.html
access: September 2014

[16] E. Burmako, M. Odersky *Scala Macros, a Technical Report*. École Polytechnique Fédérale de Lausanne (EPFL), Lipiec 2012

[17] Y. Coppel *Reflecting Scala*. Laboratory for programming methods, EPFL, Styczeń 2008

[18] E. Burmako *Scala Macros: Let Our Powers Combine!* EPFL, Lipiec 2013