Type-Safe Regular Expressions

Olivier Blanvillain
École Polytechnique Fédérale de Lausanne
Switzerland
olivier.blanvillain@epfl.ch

Abstract

Regular expressions can easily go wrong. Capturing groups, in particular, require meticulous care to avoid running into off-by-one errors and null pointer exceptions. In this chapter, we propose a new design for Scala's regular expressions which completely eliminates this class of errors. Our design makes extensive use of match types, Scala's new feature for type-level programming, to statically analyze regular expressions during type checking. We show that our approach has a minor impact on compilation times, which makes it suitable for practical use.

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Domain specific languages**; **Data types and structures**.

Keywords: regular expressions, type safety, match types

ACM Reference Format:
Olivier Blanvillain. 2018. Type-Safe Regular Expressions. In *Proceedings of Proceedings of the 13th ACM SIGPLAN Scala Symposium (SCALA'22).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Capturing groups allow programmers to extract substrings matched by parts of a regular expression. For example, the regex `"A(B)?"` matches both `"A"` and `"AB"`. In the first case, the capturing group is empty, and in the second case, the capturing group contains `"B"`.

The Scala standard library contains a package to manipulate regular expressions. This package is based on Java's implementation, and thus benefits from the high performance of Java's regex engine. The Scala implementation innovates in its presentation: it improves upon Java's solution by providing an ergonomic API based on pattern matching that is both elegant and concise. The package's documentation starts with the following example:

```
val date = Regex("""(\d{4})-(\d{2})-(\d{2})""")
"2004-01-20" match
  case date(y, m, d) =>
    s"$y was a good year for PLs."
```

The extractor pattern, `case date(y,m,d)`, replaces the need for manually indexing into the regular expression's capturing groups, and shields users from off-by-one error.

While the syntax used in this example would undoubtedly make some non-Scala programmers envious, its type safety leaves much to be desired. First of all, the *number* of variable bindings in the extractor is entirely opaque to Scala's type system, and left to the discretion of the programmer. Furthermore, the values that come out of capturing groups can be null (when using optional captures) and thus require an additional layer of validation. Those shortcomings might appear benign on small examples, but can easily turn into bugs when dealing with a large-scale codebase.

In this paper, we propose a new design for Scala's regular expression library, which provides a type-safe and null-safe mechanism for capturing group extraction. Our design makes extensive use of match types [2], Scala 3's new feature for type-level programming, to statically analyze regular expressions during type checking. We build our interface to mimic Scala's original regular expression API, so that Scala programmers can use it as a drop-in replacement and enjoy the additional safety with minimal migration costs.

This paper is structured as follows. In Section 2, we give an introduction to match types and generic tuples, two recent additions to Scala's type system which we rely on in our implementation. In Section 3, 4 and 5, we present our library for type-safe regular expressions. In Section 6, we evaluate the performance of our implementation in terms of compilation times and execution times. In Section 7, we discuss related work. We conclude the paper in Section 8.

The source code of our implementation is available online[1], under the MIT license.

2 Background

In this section, we give a brief introduction to two recent additions to Scala's type system: match types and generic tuples. Our regular expression library makes extensive use

SCALA'22, June 06, 2022, Berlin, Germany

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1] https://github.com/OlivierBlanvillain/regsafe

of those two features to enable type-safe and null-safe capturing group extraction.

2.1 Match Types

Match types provide first-class support for type-level computations in the form of pattern matching on types:

```
type Elem[X] = X match
  case String => Char
  case Array[t] => Elem[t]
  case Any => X
```

This example defines a type `Elem` parametrized by one type parameter `X`. The right-hand side is defined in terms of a match on that type parameter – a match type. A match type reduces to one of its right-hand sides, depending on the type of its scrutinee. For example, the above type reduces as follows:

```
Elem[String] =:= Char
Elem[Int] =:= Int
Elem[Array[Int]] =:= Int
```

To reduce a match type, the scrutinee is compared to each pattern, one after the other, using subtyping. For example, although `String` is a subtype of both `String` and `Any`, the `Elem[String]` type reduces to `Char` because the corresponding case appears first.

2.2 Generic Tuples

Scala's tuples were originally defined as plain old data types. In Scala 3, tuples got enhanced with a generic representation [1], similar to heterogeneous lists [5]. This representation uses two types, `EmptyTuple` and `*:`, to describe tuples in a list-like fashion. The compiler treats those new types and the traditional class-based representation of tuples interchangeably. A 2-tuple of integers, for example, has two equivalent representations:

```
(Int, Int) =:= Int *: Int *: EmptyTuple
```

This new tuple representation is especially useful at the type level, where it allows programmers to manipulate tuples recursively. For example, the following match type reverses the order of a tuple's elements:

```
type Reverse[T <: Tuple] = Rev[T, EmptyTuple]

type Rev[T <: Tuple, Acc <: Tuple] <: Tuple =
  T match
    case x *: xs => Rev[xs, x *: Acc]
    case EmptyTuple => Acc
```

3 Architecture

We present our library for type-safe regular expressions in terms of 3 components:

1. The type-level capturing group analysis, which uses match types to inspect the user-provided regular expression and compute tuple representation of the expression's capturing groups. This component takes the form of a parametric type called `Compile`, which we present in Section 4.
2. The runtime capturing group processing component, which extracts and sanitizes the output of Java's regex engine, in accordance with the previously computed type-level representation. This component takes the form of a function called `transform`, which we present in Section 5.
3. The user interface, which ties everything together to compile, execute, and extract the results of a regular expression, while providing a type-safe and null-safe experience.

We begin our presentation with the user interface, which takes the form of a relatively simple Scala class and companion object:

```
object Regex:
  def apply[R <: String & Singleton](regex: R) =
    new Regex[Compile[R]](regex)

class Regex[P] private (val regex: String):
  val pattern = Pattern.compile(regex)
  def unapply(s: String): Option[P] =
    val m = pattern.matcher(s)
    if (m.matches())
      val a = Array.tabulate(m.groupCount)(
        i => m.group(i + 1))
      Some(transform[P](regex, a))
    else
      None
```

The `apply` method is the entry point of our library. It takes a regular expression and returns an instance of `Regex`. Instead of directly accepting a `String` argument, that method takes a type parameter, `R`, which will be inferred by the compiler, and allows us to get our hands on a type-level instance of the regular expression's `String`[2]. We use that type-level string to instantiate the `Compile` type, which is our type-level analysis component.

The implementation of the `Regex` class is straightforward. Similar to its standard library counterpart, that class uses an instance of `java.util.regex.Pattern` to match the given input against the given regular expression, and to extract the content of the capturing groups when the matching succeeds. These operations happen within the `unapply` method,

[2]The `Singleton` bound is a marker that instructs type inference to preserve unions and literal singleton types. By default, the compiler widens those types, notably in method applications. For instance, given `1 :: Nil`, Scala's default type inference yields `List[Int]` instead of the more precise `List[1]`. To alter this behavior, one could mark `::`'s type argument with the `Singleton` bound.

which is the way to define custom pattern matching extractors in Scala [3].

The main difference between Scala's original `Regex` implementation and ours lies in the signature of `unapply`, which determines the nature of pattern matching extraction performed by that method. In our implementation, we instantiate the type `P` to a tuple and use that type in the result type of `unapply` to tell the compiler that a successful matching consists of exactly n capturing groups, where n is the tuple's arity. The `P` type is composed of `String` and `Option[String]` types, depending on the nullability of the correspond capturing group. Our implementation enriches the results of Java's regex engine by wrapping nullable values into options, and packaging the overall result into a tuple of type P. We perform this operation in the `transform` method, which is the runtime component of our library. As a result, our implementation never returns null values and removes the null-checking burden from the user.

For instance, in the following example usage of our library, we use a regular expression with an optional capturing group to extract the integral and fractional parts of a rational number:

```
val rational = Regex("""(\d+)(?:\.(\d+))?""")
"3.1415" match
  case rational(i, Some(f)) =>
    val n = i.size + f.size
    s"This number is $n digits long"
```

Here, we instantiate `Regex`'s type parameter to `P = (String, Option[String])`, which we then use in `unapply`'s result type to obtain a precise representation of the regex's capturing groups. The resulting code is type-safe: if we change the pattern to omit the option unpacking (replacing `Some(f)` by `f`), the compiler would raise a type error on the call to `.size` (size is defined on strings but not options).

In the following section, we explain our technique to analyze regular expressions and statically compute a type-level representation of the regex's capturing groups (`P` in the example above).

4 Type-Level

The purpose of our type-level component is two-fold:

1. Identify the capturing groups of a regular expression.
2. Determine which of those capturing groups are optional, that is, whether or not the regex engine could possibly assign null values for each of those groups.

The result of that type-level computation takes the form of a Scala tuple with `String` and `Option[String]` elements, depending on the nullability analysis.

We present our implementation incrementally, starting from a simple but incomplete solution, and progressively building up towards our final solution.

4.1 Capturing Group Identification

The first version of our type-level program is limited to capturing group identification. The implementation is straightforward, it iterates through the regex's characters and accumulates a `String` type for every opening parenthesis:

```
import compiletime.ops.string._
import compiletime.ops.int.+

type Compile[R <: String] =
  Reverse[Loop[R, 0, Length[R], EmptyTuple]]

type Loop[R, Lo, Hi, Acc <: Tuple] =
  Lo match
    case Hi => Acc
    case _  => CharAt[R, Lo] match
      case '('  => Loop[R, Lo + 1, Hi, String *: Acc]
      case '\\' => Loop[R, Lo + 2, Hi, Acc]
      case _ => Loop[R, Lo + 1, Hi, Acc]
```

The `Length` and `CharAt` types (from `ops.string`) are special-cased by the compiler: when their first argument is a known string literal, the compiler evaluates those types via their corresponding term-level implementation. Similarly, the `+` type (from `ops.int`) allows us to manipulate integers just like we would at the term level. The `EmptyTuple` and `*:` types are Scala 3's new generic representation of tuples, presented in Section 2.2.

4.2 Out-Of-Bound Errors

Attentive readers might have noticed that our handling of backslash, regex's escape character, might result in off-by-one errors. Indeed, the `Loop` type terminates when `Lo = Hi`, but increases `Lo` by 2 to skip over escaped characters.

Rest assured, this is not an oversight! Regular expressions with trailing backslashes are invalid and result in runtime crashes (reported as an "unexpected internal error", on the JVM). If we invoke `Compile` on a regular expression with a trailing backslash, the compiler will run into a call to `charAt` with an out of bounds argument, catch the corresponding exception and report it as a compilation error, which is certainly better reporting that same error at run-time. At the time of writing, the Scala 3 compiler does not support the customization of match type errors. The systematic detection of invalid regular expressions is out of the scope of this paper.

4.3 Non-Capturing Groups

Our first implementation of `Compile` treats every opening parenthesis as the start of a capturing group, which does not honor the syntax of Java's regular expression. Indeed, Java also supports several other special constructs that start with an opening parenthesis, for instance:

- non-capturing groups, `(?:X)`,

- lookaheads, `(?=X)` and `(?!X)`,
- lookbehinds, `(?<=X)` and `(?<!X)`.

To correctly identify capturing groups (which can be either named, `(?<name>X)`, or unnamed, `(X)`), we must differentiate them from other special constructs. Our second implementation (omitted) uses the following `IsCapturing` predicate type to rule out non-capturing groups:

```
type IsCapturing[R <: String, At <: Int] =
  CharAt[R, At] match
    case "?" => CharAt[R, At + 1] match
      case "<" => CharAt[R, At + 2] match
        case "=" | "!" => false // lookbehinds
        case _ => true // named-capturing group
      case _ => false // other special constructs
    case _ => true // unnamed-capturing group
```

Similar to our handling of backslash characters, this implementation intentionally does not prevent out-of-bounds errors, since these errors correspond to ill-formed regular expressions.

4.4 Nullability Analysis

Establishing the nullability of capturing groups is more complicated than it sounds. At first glance, it seems that this is simply a matter of looking for regex quantifiers in suffix position[3]. For instance, in the following naive implementation of `IsNullable`, we look for the first closing parenthesis and inspect the following characters to determine if a capturing group is nullable:

```
type IsNullable[R <: String, At, Hi] =
  CharAt[R, At] match
    case ')'  => IsMarked[R, At + 1, Hi]
    case '\\' => IsNullable[R, At + 2, Hi]
    case _    => IsNullable[R, At + 1, Hi]

type IsMarked[R <: String, At, Hi] =
  At match
    case Hi => false
    case _ => CharAt[R, At] match
      case '?' | '*' => true
      case _ => false
```

Unfortunately, this naive solution does not handle regular expressions with nested capturing groups. For example, in `"(A(B)?)"`, that solution incorrectly labels the first group as optional. To overcome this problem, we update our solution to keep track of the number of opening and closing parentheses (`Lvl`), which allows us to differentiate closing parentheses of inner and outer groups[4]:

```
type IsNullable[R <: String, At, Hi, Lvl <: Int] =
  CharAt[R, At] match
    case ')' => Lvl match
      case 0 => IsMarked[R, At + 1, Hi]
      case _ => IsNullable[R, At + 1, Hi, Lvl - 1]
    case '(' => IsNullable[R, At + 1, Hi, Lvl + 1]
    case '\\' => IsNullable[R, At + 2, Hi, Lvl]
    case _    => IsNullable[R, At + 1, Hi, Lvl]
```

Nested capturing groups bring another complication to the nullability analysis, which is caused by the interaction between inner and outer groups. When an outer group is deemed optional, this overrides the nullability of all its inner groups, which also become optional. For instance, in `"(A(B))?"`, both the first (`A(B)`) and the second (`B`) capturing group are optional.

We handle nested groups by updating our algorithm to operate in two different modes while iterating through the regex's characters. Our algorithm can either be outside of an optional group (`Opt=0`), in which case it computes nullability of newly encountered groups via `IsNullable`, or inside an optional group (`Opt>0`), in which case it treats every group as nullable.

With this latest improvement, we advance our implementation to its final iteration (we omit usages of `IsCapturing`, from Section 4.3, for brevity):

```
type Loop[R, Lo, Hi, Opt <: Int, Acc <: Tuple] =
  Lo match
    case Hi => Acc
    case _  => CharAt[R, Lo] match
      case ')' =>
        Loop[R, Lo+1, Hi, Acc, Max[0, Opt-1]]
      case '(' => Opt match
        case 0 => IsNullable[R, Lo+1, Hi, 0] match
          case true =>
            Loop[R, Lo+1, Hi, Option[String]*:Acc, 1]
          case false =>
            Loop[R, Lo+1, Hi, String*:Acc, 0]
        case _ => Loop[R, Lo+1, Hi,
                       Option[String]*:Acc, Opt+1]
      case '\\' => Loop[R, Lo+2, Hi, Opt, Acc]
      case _ => Loop[R, Lo+1, Hi, Opt, Acc]
```

This concludes the development of our nullability analysis and completes our presentation of the type-level component of our library. Despite the scarcity of Scala's standard library at the type level, we managed to achieve our ends while keeping our implementation relatively concise and,

[3]For brevity, our presentation does not account for the regex alternative operator, which also influences the nullability of capturing groups and can appear in both prefix and suffix positions. Similarly, we omit handling the "at least n times" quantifiers which can lead to nullable capturing groups (when $n = 0$). Our implementation accounts for both operators.

[4]For simplicity, our presentation treats all non-escaped parentheses as capturing group delimiters. Our implementation also takes `\Q...\E` quotations into account and ignores parentheses that appear in character classes.

hopefully, understandable. In its final iteration, our capturing group analysis is, to the best of our knowledge, on par with Java's implementation of regular expression.

5 Term-Level

The runtime component of our library is in charge of sanitizing and packaging the results of regular expression matchings. Concretely, this component's job consists of transforming the string array that comes out of the regex engine into an appropriately sized tuple and wrapping the nullable elements in options. That transformation must conform to the representation computed at the type level.

In Scala, all type parameters are erased, which leads to some friction when wanting to write programs whose execution *depends* on types, such as in the problem at hand. In this section, we propose two different solutions to that problem.

1. In Section 5.1, we proceed by sheer force of code duplication: we translate the entirety of our type-level algorithm into term-level functions and use a cast to correlate the two.
2. In Section 5.2, we show how to use implicits to perform type-directed code synthesis: we use the output of our type-level analysis as an input of implicit search to generate a type-specialized capturing group sanitizer.

5.1 We Don't Need No Dependent Types!

The first implementation of our runtime component duplicates the type-level definitions presented in the previous section to compute a list of sanitizing functions that correspond to the given regex's capturing groups. In a nutshell, instead of accumulating `String` and `Option[String]` in a type, we accumulate identity functions and eta-expansions of the option constructor in a list:

```
val id: String => Any = { x => assert(x != null); x }
val nu: String => Any = { x => Option(x) }

def loop(r: String, lo: Int, hi: Int, acc: List[
    String => Any], opt: Int): List[String => Any] =
  lo match
    case `hi` => acc
    case _  => r.charAt(lo) match
      case ')' =>
        loop(r, lo+1, hi, acc, 0.max(opt-1))
      case '(' => opt match
        case 0 => isNullable(r, lo+1, hi, 0) match
          case true =>
            loop(r, lo+1, hi, nu::acc, 1)
          case false =>
            loop(r, lo+1, hi, id::acc, 0)
        case _ => loop(r, lo+1, hi, nu::acc, opt+1)
      case '\\' => loop(r, lo+2, hi, acc, opt)
      case _ => loop(r, lo+1, hi, acc, opt)
```

After having computed those functions, we do a pointwise application with the raw output of the regex engine, package the result into a tuple, *et voilà!*

```
def transform[P](r: String, arr: Array[String]): P =
  val fs = loop(r, 0, r.length, Nil, 0).reverse
  val wrapped = arr.zip(fs).map { (x, f) => f(x) }
  val tuple = Tuple.fromArray(wrapped)
  assert(arr.size == fs.size)
  tuple.asInstanceOf[P]
```

This solution has the benefit of being conceptually simple. We needed to write an implementation that conforms to our type-level program, and this is literally what we did, by duplicating that program and using an unsafe cast to convince the compiler of our good intentions.

The first alternative that comes to mind is perhaps to turn ourselves to a dependently typed language with support for type- and term-level polymorphism. Using a language with shared term and type syntax would allow us to write our analysis generically, and derive two programs from it, one for each level.

In principle, Scala 3's type inference should be able to solve half of that problem with its ability to correlate match types and match expressions [2, § 2.1].In our case, this mechanism should allow us to get rid of the unsafe cast but does not address the problem of code duplication. In practice, at the time of writing, this solution does not play well with predefined types from the standard library: the compiler lacks the knowledge necessary to correlate predefined types with their term-level counterpart. For instance, the compiler does not recognize `CharAt["hello", 0]` as a valid type for the `"hello".charAt(0)` expression.

In the long run, it would be interesting to investigate extending Scala with a "precise" mode of type inference, that would automatically infer match types and singleton types, whenever possible.

5.2 Implicit-Based Extractor Synthesis

The second version of our runtime component takes a completely different approach. Instead of analyzing regular expressions at run-time, we use implicit resolution to *synthesize* a type-specialized runtime, based on the results of our type-level analysis. That synthesis is type-directed: it generates a single-purpose program that sanitizes the output of the regex engine by following the shape of the tuple type computed at the type level.

Our implementation consists of one type class, `Sanitizer`, and three implicit definitions, which will guide the compiler into generating the correct `Sanitizer[T]`, for any tuple `T`

with `String` and `Option[String]` elements (here, `T` is the result of our type-level analysis). The `Sanitizer` type class defines a single method that mutates an array in place to put nullable elements in options:

```
abstract class Sanitizer[T](val i: Int):
  def mutate(arr: Array[Any]): Unit

object Sanitizer:
  implicit val basecase =
    new Sanitizer[EmptyTuple](0):
      def mutate(arr: Array[Any]): Unit = ()
  implicit def stringcase[T <: Tuple](implicit ev:
    Sanitizer[T]) =
      new Sanitizer[String *: T](ev.i+1):
        def mutate(arr: Array[Any]): Unit =
          assert(arr(arr.size-i) != null)
          ev.mutate(arr)
  implicit def optioncase[T <: Tuple](implicit ev:
    Sanitizer[T]) =
      new Sanitizer[Option[String] *: T](ev.i+1):
        def mutate(arr: Array[Any]): Unit =
          arr(arr.size-i) = Option(arr(arr.size-i))
          ev.mutate(arr)
```

From a design standpoint, this solution is appealing as it leads to no duplicated code or computation: we perform our capturing group analysis at the type-level, once and for all, and synthesize code accordingly. This combination of type-level programming and implicit-based code synthesis is reminiscent of multi-stage metaprogramming, in its compile-time variant [12]. In those terms, our approach corresponds to a multi-staged program whose stage 0 happens entirely within Scala's type checker.

In the following section, we evaluate the performance of our library and discuss the trade-offs between the first and second implementation of our runtime component.

6 Evaluation

To assess the correctness of our library, we run our implementation against the QT3TS test suite [14], which consists of about 1700 test cases for regular expressions [15]. We select the subset of those tests that are positive and exercise capturing groups, totaling 183 test cases, which we then collated to form our benchmark suite. We use these benchmarks to evaluate the performance of our library in terms of compilation times and execution times.

In Figure 1, we show the results of our experiments, which compare Scala's standard regular expression library (Std) against two flavors of our library, one using our code-duplicated runtime (Dup), presented in Section 5.1, and the other using our implicit-based runtime (Impl), presented in Section 5.2.

We obtained these results by averaging the measurements obtained over 2-hour runs, for each data point, which we executed on an i7-7700K Processor running Oracle JVM 1.8.0 and Linux. With these settings, we obtain margins of errors below ±1%, with 99.9% confidence (omitted in Figure 1). We run our benchmarks on the latest version of Scala 3 at the time of writing (3.1.2).

Compilation Times. Our library in its code-duplicated variant adds a total of 53 ms over the baseline's total compilation time (an average of +0.3 ms per regex). This corresponds to the cost of match type reduction. The implicit-based variant adds another 21 ms (+0.1 ms per regex), which corresponds to the cost of implicit resolution. Both variants are thus relatively cheap, which validates their utility for practical uses. When it comes to compilation times, it would be misleading to compare increments relative to the overall time, given that the compiler spends a large portion of its time outside of type checking [8, § 2.11.3].

Execution Times. At runtime, our code-duplicated variant adds 51 µs over the baseline. Although this difference is three orders of magnitude smaller than at compile-time, it corresponds to a 25% increase, which is a rather substantial price to pay for redoing an already-performed analysis (`transform` redoes the analysis performed by `Compile`). The implicit-based variant gets rid of this cost by synthesizing a type-specialized runtime (at the cost of increased compilation time). To our surprise, our implicit-based runtime outperforms Scala's standard library implementation. We suspect that this difference is due to the poor interaction between the JVM's just-in-time compiler and the code generated for variadic extractors (`unapplySeq`, used by the Std implementation). Our implicit-based implementation appears to be more prone to inlining and partial evaluation since it does not involve generic sequences, which could explain the slight gain in performance.

In view of these results, we decided to use our implicit-based runtime in the published version of our library.

7 Related Work

We found surprisingly little literature concerned with the safety of regular expression capturing groups. In a recent survey paper on regular expression correctness, Zheng et al. [17] classified the publications in this area into several categories, ranging from empirical studies to regular expression synthesis and repair. However, they only listed a single paper about static checking: the type system developed by Spishak et al. [11] to validate regular expression syntax and capturing group usage in Java programs. Their tool takes the form of a compiler plugin that enhances Java's type system to track the number of capturing groups using type

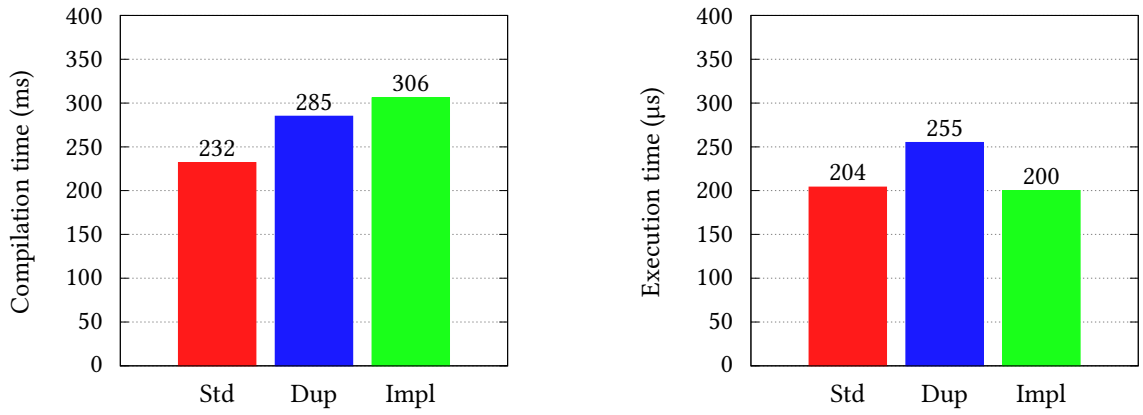

Figure 1. A comparison of the compilation time (left) and execution time (right) of Scala's standard regex library (Std) against our library with its code-duplicated runtime (Dup) and with its implicit-based runtime (Impl).

annotations. Unlike our system, their solution is not concerned with null safety and does not detect optional capturing groups. In practice, their tool operates similarly to a linter: it reports false positives and sometimes requires manual annotations.

The Haskell ecosystem contains several packages for regular expressions, with various levels of static safety. The regex-applicative package stands on the safe side of that spectrum: it allows programmers to write regular expressions using a parser combinator library, which entirely removes the need to reference capturing groups [10]. Tyre [9] is an OCaml library that also uses combinators to define type-safe regular expressions. The main drawback of these solutions is the learning curve: programmers already familiar with POSIX regular expressions [4] are forced to learn a new syntax, which is arguably more complicated.

Stephanie Weirich presented on several occasions a regular expression library as an example of a dependently-typed program written in Haskell [16]. The library provides safety guarantees similar to ours, but from the slightly different angle of named-capturing groups. An important conceptual difference between the two approaches is that ours only focuses on type checking (we rely on Java's regular expressions at run-time), whereas hers also includes a fully-fledged regular expression engine.

The typed-regex [7] Typescript library provides a type-safe API for extracting named capturing groups of regular expressions. The implementation uses conditional types [13], Typescript's type-level ternary operator, to statically analyze regular expressions. Their approach, like ours, identifies optional capturing groups and marks them as such (using Typescript's optional properties). At the time of writing, the library is still in the early stages of development and only supports a subset of the regular expression syntax, and notably lacks support for nested optional capturing group detection.

The safe-regex [6] Rust library takes a different approach towards the same goal: it uses Rust macros to statically compile regular expressions and generate arity-specific extractors. Performance-wise, macros improve over the status quo since they remove the need to compile regular expressions at runtime. Safety-wise, code generation provides a straightforward solution to safe capturing group extraction.

8 Conclusion

In this paper, we introduce a new design for type-safe regular expressions in Scala. We presented our type-level analysis, which identifies capturing groups and computes their nullability. We built our library following the design of the original Scala regular expression API, in order to provide a frictionless migration path for programmers interested in the additional type safety. We evaluated our implementation by running it against the QT3TS test suite and showed that, on those benchmarks, our type-level analysis only has a marginal impact on compilation times.

Future Work. We propose to extend our approach with a dedicated data type for the regex alternative operator. In our current design, we map every capturing group to an `Option[String]`, which does not always accurately reflect the structure of regular expressions. For instance, when confronted with `"(A)(B)|(C)"`, our system represents this expression's capturing groups as a 3-tuple, `(Option[String], Option[String], Option[String])`. This representation fails to account for the fact that the first two groups have identical nullability status, and that this status is opposite to the nullability of the third group. Instead, we could represent those capturing groups as `Either[(String, String),`

`String`], which is isomorphic to the structure of the regular expression.

References

[1] Vincenzo Bazzucchi. 2021. *Tuples Bring Generic Programming to Scala 3.* https://www.scala-lang.org/2021/02/26/tuples-bring-generic-programming-to-scala-3.html.

[2] Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. 2022. Type-Level Programming with Match Types, In Proc. ACM Program. Lang. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages.* https://doi.org/10.1145/3498698.

[3] Burak Emir, Martin Odersky, and John Williams. 2007. Matching Objects with Patterns. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'07).* Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-73589-2_14.

[4] IEEE. 2018. *The Open Group Base Specifications Issue 7, 2018 edition.* https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html.

[5] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell'04).* ACM, New York, NY, USA. https://doi.org/10.1145/1017472.1017488.

[6] Michael Leonhard. 2021–2022. safe-regex GitLab Repository. https://gitlab.com/leonhard-llc/safe-regex-rs.

[7] Akshay Nair. 2021. typed-regex GitHub Repository. https://github.com/phenax/typed-regex.

[8] Dmytro Petrashko. 2017. *Design and implementation of an optimizing type-centric compiler for a high-level language.* PhD dissertation. EPFL, Lausanne. https://doi.org/10.5075/epfl-thesis-7979.

[9] Gabriel Radanne. 2017–2020. Tyre GitHub Repository. https://github.com/Drup/tyre.

[10] Cheplyaka Roman. 2011–2021. regex-applicative. https://github.com/UnkindPartition/regex-applicative.

[11] Eric Spishak, Werner Dietl, and Michael D. Ernst. 2012. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs (FTfJP'12).* ACM, New York, NY, USA. https://doi.org/10.1145/2318202.2318207.

[12] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'18).* ACM, New York, NY, USA. https://doi.org/10.1145/3278122.3278139.

[13] The TypeScript development team. 2019–2022. *The TypeScript Handbook.* Microsoft Corporation. https://www.typescriptlang.org/docs/handbook/intro.html.

[14] W3C. 1994–2013. XQuery/XPath/XSLT 3.* Test Suite (QT3TS). https://dev.w3.org/2011/QT3-test-suite/.

[15] W3C. 2021. QT3TS GitHub Repository. https://github.com/w3c/qt3tests.

[16] Stephanie Weirich. 2014–2020. Examples of Dependently-typed programs in Haskell. https://github.com/sweirich/dth.

[17] Li-Xiao Zheng, Shuai Ma, Zu-Xi Chen, and Xiang-Yu Luo. 2021. Ensuring the Correctness of Regular Expressions: A Review. *International Journal of Automation and Computing* 18, 4 (2021). https://doi.org/10.1007/s11633-021-1301-4.