Strongly-typed regular expressions in Dotty

Bachelor Semester Project Spring 2019 Final Report

Andrea Veneziano
EPFL
andrea.veneziano@epfl.ch

Abstract

With the goal of analyzing two new Dotty features, dependent and match types, we present a strongly-typed library for regular expressions. Developed for type-level programming as a way to provide additional type safety, these experimental functionalities are here used to improve the expressiveness of the Scala standard library. With our implementation, we demonstrate that dependent and match types can achieve their purpose, although the user might forego some performance and ease of use.

1. Introduction

In today's world, we witness an increasingly need for type safety. Software is more complex than ever, and programmers want to be able to create interfaces with enhanced static information [1]. The goal is to increase the reliability and security of programs by specifying and enforcing rich data invariants [12]. In other words, we seek domain-specific type checkers that could help us catching more errors at compile time, instead of waiting for them to surface at run-time.

With this goal in mind, the expressiveness of the type system can be increased by enabling type-level programming. In recent years, there has been considerable interest in the research community for this topic, and two solutions have mainly stood up: dependent types and type families.

Dependent types are types whose definition depends on a value. They can be used to maintain a data structure's type as precise as its implementation. They have been widely adopted in proof assistants such as Agda [10], Coq [4], Epigram [8] and Idris [3]. Although almost absent from general-purpose programming languages, they are starting to be partially adopted in Haskell [6], JavaScript [5] and Scala's next generation compiler, Dotty [2].

Type families are parametric types that can be assigned specialized representations based on the type parameters they are instantiated with. Being functions on types, they can be used, for example, to dynamically transform a function's return type based on its arguments' types. Match types [11], under active development, are, with few differences, Dotty's version of Haskell closed type families [7]; they share some similarities with TypeScript's conditional types too [9].

In this work, we analyze dependent and match types, Dotty's new features, by asserting their usability, both in terms of programming experience and performance, in the context of the development of a strongly-typed library for regular expressions. Inspired by Weirich's talk [13], our library has, at compile time, full knowledge of the types of the capturing groups inside a regex. For example, the simple regex "[0-9]", once compiled, has type *String => Option[Int]*. This allows the type system to provide stronger guarantees to the programmer, which can specify the type of the desired regex and exploit the additional safety, given by the type checker, at compile time.

This is the first work, to our knowledge, to use Dotty's match and dependent types, still under development, in a relatively complex program. It is our hope that the present report will help Dotty developers' task and serve as a reference for future users.

The rest of the report is organised as follows. In Section 2, we illustrate with a concrete example from the library how these new technologies work. In Section 3, we describe in details the library implementation, showing how these features can be used. In Section 4, we present two possible use cases that demonstrate how dependent and match types can improve type safety. In Section 5, we compare the two implementations, highlighting strengths and weaknesses. In Section 6, we conclude with a description of possible extensions to our work.

The source code for the regex library is available online[1].

2. How dependent and match types work

In this section, we present the complete implementation of the methods used to check, at compile time, whether delimiters are balanced. This code, available in Listings 1 and 2, gives us the opportunity to show how dependent and match types work.

2.1 Dependent types

Dependent types, as stated before, allow programmers to type values, methods and case classes as precisely as their implementations. In Dotty, the compiler can be instructed to do so by providing, for example, the body of a dependently-typed function inside brackets {...} as the explicit result type:

```
// the return type is 5, more precise than
// Int
def digit: { 2 + 3 } = 2 + 3
```

To avoid duplicating code, the **dependent** qualifier has been introduced; it allows the user to omit the type and let the compiler infer it:

```
// the return type is still 5
dependent def digit = 2 + 3
```

In Listing 1, we use **dependent** to precisely type the methods *checkParens()*, *checkBrackets()* and *check()*, such as the actual return type is either true or false, instead of simply Boolean. This allows us to know whether delimiters in a string are balanced directly at compile time.

More concretely, the auxiliary method *check()* reads the string one character at a time, counting opening delimiters. Then, when a closing one is found, the function checks, through the counter, if an open delimiter to match has been already encountered. If it is not the case, then the delimiters are not balanced. They are balanced only if we finish traversing the string and the counter is equal to zero.

2.2 Match types

Match types can be viewed as functions on types defined through pattern matching. They reduce, depending on a scrutinee type, to one of several candidates, expressed as right hand sides:

```
type IsDigit[X] = X match {
    case 2 => true
    case 'a' => false
    ...
}

val digit: IsDigit[2] = true
val character: IsDigit['a'] = false
```

The *digit*'s type is true, since IsDigit[2] gets transformed to true, while the *character*'s type is false.

In Listing 2, we use match types to specialize, based on a given string, *CheckParens*, *CheckBrackets* and *Check*, in such a way that they evaluate to either true or false. This allows us to know whether delimiters in a string are balanced at compile time.

The *Check* implementation is a bit more involved than its dependent counterpart, as the counter is defined using type-level natural numbers. The logic is the same, but additional checks are needed to simplify the interactions between *Pred[x]* and *Suc[x]*, which represent, respectively, the predecessor and the successor of a number. Indeed, both *Pred[Suc[x]]* and *Suc[Pred[x]]* are equivalent to *x*.

[1] https://github.com/Gondolav/dotty/tree/match-types-regex/strongly-typed-regex

```
dependent def checkParens(s: LstChar): Boolean = check(s, 0, '(', ')')

dependent def checkBrackets(s: LstChar): Boolean = check(s, 0, '[', ']')

dependent private def check(cs: LstChar, opened: Int, open: Char, close: Char): Boolean =
cs match {
   case Nil => opened == 0
   case open :: xs => check(xs, opened + 1, open, close)
   case close :: xs if (opened < 1) => false
   case close :: xs => check(xs, opened - 1, open, close)
   case x :: xs => check(xs, opened, open, close)
}

// example: if we write "false" instead of "true", the program will not compile
val balanced: true = checkParens(Cons('(', Cons('(', Cons(')', Cons(')', Nil)))))
```

Listing 1. Dependent types implementation of a delimiter balance checker.

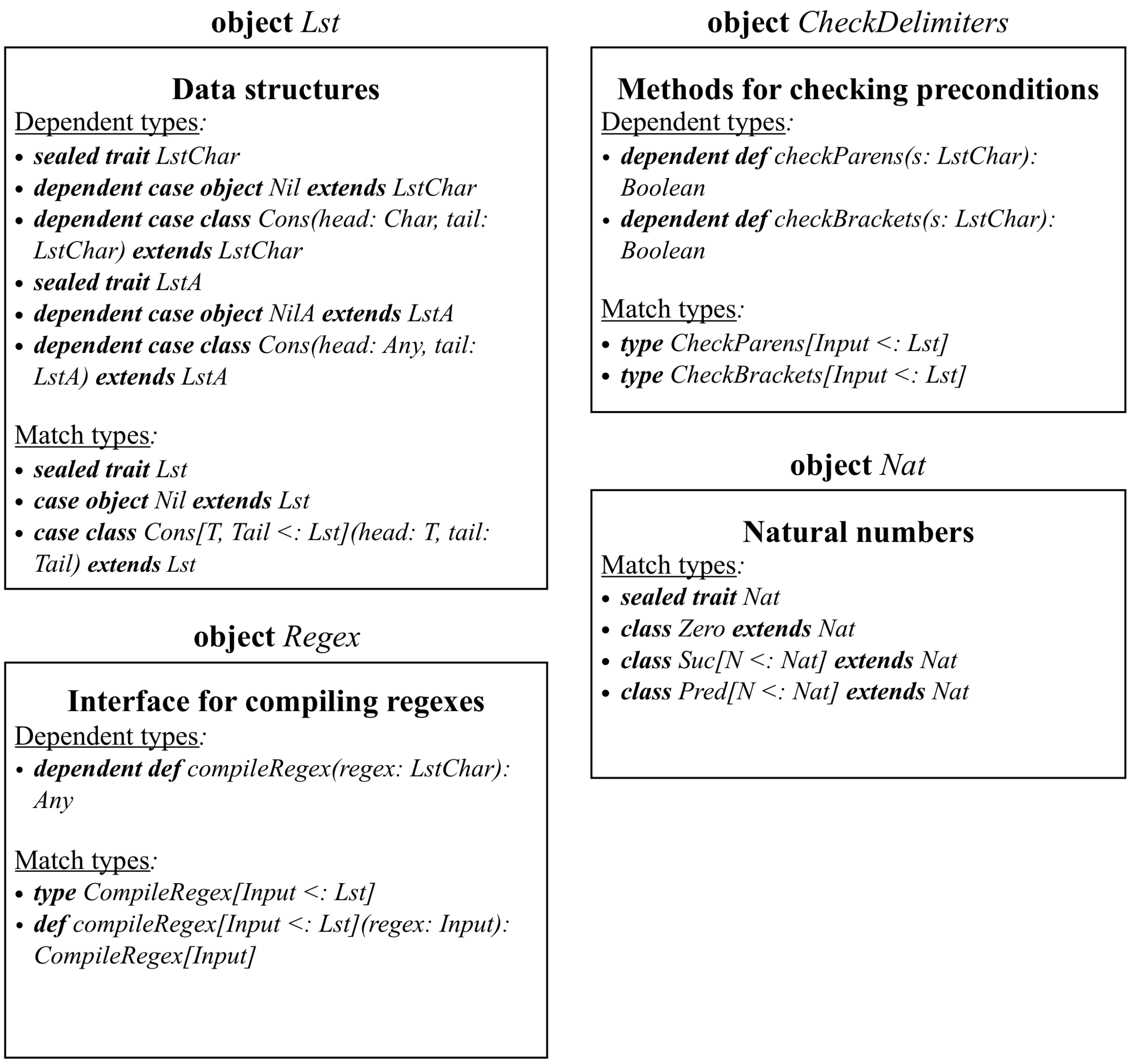
```
type CheckParens[Input <: Lst] = Check[Input, Zero, '(', ')']

type CheckBrackets[Input <: Lst] = Check[Input, Zero, '[', ']']

type Check[Input <: Lst, Opened <: Nat, Open <: Char, Close <: Char] = Input match {
   case Nil.type => Opened match {
      case Zero => true
      case _ => false
   }
   case Cons[Open, xs] => Opened match {
      case Pred[o] => Check[xs, o, Open, Close]
      case _ => Check[xs, Suc[Opened], Open, Close]
   }
   case Cons[Close, xs] => Opened match {
      case Zero => false
      case _ => Opened match {
         case Suc[o] => Check[xs, o, Open, Close]
         case _ => Check[xs, Pred[Opened], Open, Close]
      }
   }
   case Cons[_, xs] => Check[xs, Opened, Open, Close]
}

// example: if we write "false" instead of "true", the program will not compile
val balanced: CheckParens[Cons['(', Cons['(', Cons[')', Cons[')', Nil.type]]]]] = true
```

Listing 2. Match types implementation of a delimiter balance checker.



object *Lst*

Data structures

Dependent types:

- ***sealed trait*** *LstChar*
- ***dependent case object*** *Nil* ***extends*** *LstChar*
- ***dependent case class*** *Cons(head: Char, tail: LstChar)* ***extends*** *LstChar*
- ***sealed trait*** *LstA*
- ***dependent case object*** *NilA* ***extends*** *LstA*
- ***dependent case class*** *Cons(head: Any, tail: LstA)* ***extends*** *LstA*

Match types:

- ***sealed trait*** *Lst*
- ***case object*** *Nil* ***extends*** *Lst*
- ***case class*** *Cons[T, Tail <: Lst](head: T, tail: Tail)* ***extends*** *Lst*

object *CheckDelimiters*

Methods for checking preconditions

Dependent types:

- ***dependent def*** *checkParens(s: LstChar): Boolean*
- ***dependent def*** *checkBrackets(s: LstChar): Boolean*

Match types:

- ***type*** *CheckParens[Input <: Lst]*
- ***type*** *CheckBrackets[Input <: Lst]*

object *Regex*

Interface for compiling regexes

Dependent types:

- ***dependent def*** *compileRegex(regex: LstChar): Any*

Match types:

- ***type*** *CompileRegex[Input <: Lst]*
- ***def*** *compileRegex[Input <: Lst](regex: Input): CompileRegex[Input]*

object *Nat*

Natural numbers

Match types:

- ***sealed trait*** *Nat*
- ***class*** *Zero* ***extends*** *Nat*
- ***class*** *Suc[N <: Nat]* ***extends*** *Nat*
- ***class*** *Pred[N <: Nat]* ***extends*** *Nat*

Figure 1. The project structure described in Section 3.1.

3. Implementation

In this section, we give an overview of how we used Dotty's experimental functionalities, such as match types and dependent types, to improve the expressiveness of the Scala standard library for regular expressions.

First, we will describe the overall architecture of our extension and the approach taken for its implementation. Then, we will present, in more details, the core of our work that uses these new features, and the algorithm behind it.

3.1 Architecture

We applied a functional approach to our work, relying on recursion and accumulators and avoiding side effects. Since the project main goal is to provide more insight into the use of dependent and match types, we decided to implement only a subset of regular expressions, focusing on readability rather than performance. For this reason, the supported functionalities are limited to capturing groups, simple character classes of the form [a-z], Kleene star * and optional ?.

The structure of the code, for both implementations, can be summarized into three modules (see Figure 1): data structures for representing regexes, groups types and results (object *Lst*), methods for checking preconditions and input format validity (object *CheckDelimiters*) and the main interface used for compiling regular expressions (object *Regex*). For the implementation involving match types, we have to add to this general structure also natural numbers (object *Nat*), which are needed for type-level computations.

3.2 Algorithm

The bulk of the work is performed by a simple state machine, which consumes the regex one character at a time, iteratively building up the list containining the types corresponding to each group. This list and the Scala library for regular expressions are then used to construct a closure of the form *String => Option[Cons(type, Cons(..., Nil)]*. This closure has complete knowledge of the regex groups' types at compile time and represents the pattern used to match the regex at run-time. For example, compiling the regex "[a-z]" returns a closure of type *String => Option[Cons(Char, Nil)]*.

3.2.1 Dependent types

```
dependent def compileRegex(regex: LstChar):
    Any = {
    if (checkParens(regex)) compile(regex,
        Empty, 0, false, 0, Nil, regex)
    else RegexError
}

dependent private def compile(...,
    groupsTypesRepr: LstChar, ...): Any =
    {... buildPattern(...) ...
      ... compileCharClass(...) ...}

dependent private def compileCharClass(...,
    groupsTypesRepr: LstChar, ...): Any =
    {...}

dependent private def buildPattern(...,
    groupsTypesRepr: LstChar): String =>
    Option[{ groupsTypesRepr.toLstA }] = {
    input: String =>
    ...}.asInstanceOf[String => Option[{
    groupsTypesRepr.toLstA }]]
```

Listing 3. Dependent types prototypes.

In Listing 3, dependent types are used to keep the type of the list *groupsTypesRepr* as precise as possible, in such a way that it can then be used as return type without information loss in the function *buildPattern()*.

First, *compileRegex()* checks that the regular expression is well formatted. Then, it calls *compile()* which does most of the work; it keeps track of the type of the current group traversed (*Empty*, *Str*, *Chr* or *Integ*, defined as dependent case objects in object *Regex*), the number of characters encountered (to distinguish between String and Char), whether a character class is present at the moment and how many are defined (to disambiguate between a group containing just a single char class, of type Char, and one containing multiple classes, of type String). The regex is cached as it is needed in *buildPattern()*, where it is transformed in an actual scala.util.matching.Regex.

When a character class is encountered, *compile()* checks that the brackets are balanced, and then relinquishes control to *compileCharClass()*. This function parses the received regex, extracting the class type and eventually detecting errors (such as "[z-a]").

Finally, the function *buildPattern* builds the closure to be used as pattern at run-time. Its body makes use of the Scala standard library to match the given input, and uses the information collected in *groupsTypesRepr* to change the type of each group from the default one, String, to the actual, more informative type.

3.2.2 Match types

```
type CompileRegex[Input <: Lst] =
    CheckParens[Input] match {
       case true => Compile[Input,
           Empty.type, Zero, false, Zero,
           Nil.type, Input]
       case false => RegexError.type
}

type Compile[..., GroupsTypesRepr <: Lst,
    ...] = ... match {
    ... BuildPattern[...] ...
    ... CompileCharClass[...] ...
}

type CompileCharClass[..., GroupsTypesRepr
    <: Lst, ...] = ...

type BuildPattern[GroupsTypesRepr <: Lst] =
    String =>
    Option[ToTypesList[GroupsTypesRepr]]
```

Listing 4. Match types prototypes.

In Listing 4, match types are exploited in a similar way as dependent types. In fact, the actual implementation almost mirrors the dependent one, with the notable exception that actual Scala lists are used for the code executing at run-time (functions *compile()*, *compileCharClass()* and *buildPattern()*). For this reason, we decided to highlight in this section only the type-level implementation, as it shows how match types can be used to achieve the same purpose as dependent ones with the same algorithm and few syntax changes.

The types thus defined are used as return types for the functions of the same name (Figure 1 shows an example).

4. Use cases

In this section, we present two concrete examples that demonstrate the advantages of using strongly-typed, expressive regular expressions over the weakly-typed, string-based ones implemented in the Scala standard library.

The examples highlight the benefits of knowing precisely the types of the capturing groups at compile time and show how it is possible to use this additional information to increase type-safety and reliability.

4.1 Matching URLs

In this use case, we first create a regular expression for matching URLs. Then, we extract the information retrieved from a match, storing it in a case class *URL()*. Finally, we perform an operation on one of the fields, protocol.

4.1.1 Scala regular expressions

As illustrated in Listing 5, in the Scala standard library, if an optional expression (resulting from a *, for instance) is not present, its value will be null upon matching. Thus, assuming that it has been decided beforehand that null values will be treated as *None*, the programmer has to explicitly handle this case, through cumbersome if-conditions or using the *Option()* constructor. On the other hand, if the user, by accident, forgets to perform these checks, the program will fail with an exception at run-time.

4.1.2 Regular expressions with dependent and match types

With dependent and match types (Listing 6), however, if an optional expression is not present, its value is *None* by default and its actual type is known at compile time. This means that the programmer does not have to perform null checks, because he is protected by the compiler's type-checker. The resulting code is thus easier to write and safer to use, because errors will be caught early, at compile time.

4.2 Matching sbt compilation results

In this second example, we create a regular expression for matching sbt compilation results, and use it to extract the total compilation time.

The code is available in Listings 7 and 8.

```scala
case class URL(protocol: Option[String], hostname: String, domain: String, path: Option[String])

val regexURL: Regex = "(https?://)?(www.)?([a-z][a-z]*)(.)([a-z][a-z][a-z]*)(/[a-z]*)*".r
val url: Option[URL] = "https://www.epfl.ch/schools/ic/" match {
   // if an optional is not present, its value is null
   case regexURL(protocol, _, hostname, _, domain, path) =>
      // since the Regex extractor produces varargs, the user could use the wrong group
      Some(URL(Option(protocol), hostname, domain, Option(path)))
   case _ => None
}
// if the user forgets to do null checks, this call fails with an exception at run-time
val protocolName: String = extractProtocolName(url)
```

Listing 5. Matching URLs with Scala regex.

```scala
case class URL(protocol: Option[String], hostname: String, domain: String, path:
    Option[StarMatch[String]])

val regexURL: String => Option[Cons[Option[String], Cons[Option[String], Cons[String,
    Cons[Char, Cons[String, Cons[Option[StarMatch[String]], Nil.type]]]]]]] =
    compileRegex[...](...)
val url: Option[URL] = regexURL("https://www.epfl.ch/schools/ic/") map {
      // if an optional is not present, its value is None and its type is known at compile time
      case Cons(protocol, Cons(_, Cons(hostname, Cons(_, Cons(domain, Cons(path, _)))))) =>
          URL(protocol, hostname, domain, path)
}
val protocolName: String = extractProtocolName(url)
```

Listing 6. Matching URLs with match types.

```scala
val regexCompilationResult: Regex = "(Total time: )([0-9][0-9]*)( s, completed )([A-Z][a-z]*)(
    )([0-9][0-9]?)(, )([0-9][0-9][0-9][0-9])( )([0-9][0-9]?)(:)([0-9][0-9])(:)([0-9][0-9])(
    )([A-Z][A-Z])".r
val compilationTime: Option[Int] = "Total time: 228 s, completed May 9, 2019 7:33:37 PM" match {
   // if we call toInt on the wrong group, we get an exception at run-time
   case regexCompilationResult(_, time, _: _*) => Some(time.toInt)
   case _ => None
}
```

Listing 7. Matching sbt compilation results with Scala regex.

```scala
val regexCompilationResult: String => Option[{ ConsA(_ : String, ConsA(_ : Int, ConsA(_ :
    String, ConsA(_ : String, ConsA(_ : Char, ConsA(_ : Int, ConsA(_ : String, ConsA(_ : Int,
    ConsA(_ : Char, ConsA(_ : Int, ConsA(_ : Char, ConsA(_ : Int, ConsA(_ : Char, ConsA(_ :
    Int, ConsA(_ : Char, ConsA(_ : String, NilA)))))))))))))))) }] = compileRegex(...)
val compilationTime: Option[Int] = regexCompilationResult("Total time: 228 s, completed May 9,
    2019 7:33:37 PM") map {
   case ConsA(_, ConsA(time, _)) => time // completely safe
}
```

Listing 8. Matching sbt compilation results with dependent types.

4.2.1 Scala regular expressions

The standard library is, as stated before, weakly-typed: all values resulting from a match are represented as strings. Thus, in order to retrieve the compilation time as Int in Listing 7, we have to call *toInt()* on the expression corresponding to it. This operation is unsafe, because the programmer could use, for example, the wrong extracted group, which would lead to a run-time exception.

4.2.2 Regular expressions with dependent and match types

With our approach, there is no need to use *toInt()*, because actual groups types are computed at compile time. Hence, even if the user makes a mistake and extracts the wrong expression, the compiler will complain and catch the error.

5. Results

In this section, we compare the two methods presented in Section 3, using them as a basis for a more general analysis of Dotty's current implementation of match and dependent types.

We start by analyzing their compilation time. Then, we briefly describe our experience in developing with them. Finally, we highlight some key differences we identified.

5.1 Compile time comparison

Benchmarks suggest that, in both implementations, the additional compilation time spent doing actual type-level computation is negligible compared to the overall compilation time. Moreover, compared to the Scala standard library, our work requires slightly more time to compile.

5.2 Programming experience and ease of use

Both features, for several reasons, are challenging to use. There is a general lack of documentation and of meaningful error messages, that, combined with sporadic bugs, makes programming with them more difficult. However, we found that slightly changing our programming habits to use them was the hardest part; sometimes, it was difficult to even understand how to approach a certain problem. This was especially true for match types, because pattern matching was the only language construct at disposal for pure type-level programming.

5.2.1 Dependent types

Programming with dependent types, even though pattern matching is not supported, is in general more similar to usual programming and thus easier. Nevertheless, it requires dependently-typed data structures and functions, which means that the standard library cannot be used. Hence, the programmer has to write everything on his own, from scratch.

Since equality between objects is not yet understood by dependent types, we had to use Char to represent types. We were constrained in our implementation decisions also in other situations: closures defined in a dependent context do not work well (our usage of *zipWithIndex()*, in the function *BuildPattern()*, is an evidence) and types are not correctly inferred at run-time for the expression resulting from a regex pattern matching, and thus casts are necessary.

5.2.2 Match types

Match types are arguably harder to code with, since they compel the user to program at the type level. The resulting programs are, however, easier to read and more elegant, despite being more verbose. Their elegance, compared to the dependently-typed ones, can be appreciated because the Scala standard library and pattern matching are available for usage.

Duplicated code is the main problem with match types. The code written for a function and the corresponding type-level code are essentially the same (*Compile[]* and *compile()* are clear proof). Furthermore, upon function calls, as it is done for polymorphic methods, all types must be specified (including potentially long type-level regexes in our implementation) which can be tedious, error-prone and another source of code duplication.

Another limitation we encountered is the impossibility to perform inequality comparisons on types. This prevents error detection in character classes when the first element is greater than the second.

Finally, it is not possible to perform arithmetic operations or simple counting on integers at the type level, which led us to use type-level natural numbers instead.

6. Conclusion

In this report, Dotty's dependent types and match types have been proved capable of trading off compile time performance and ease of use for additional type safety. Here, some possible extensions and improvements to the presented regex library are described.

The current work covers a subset of regular expressions; a natural extension would be to support a larger subset, such as +, ^ (not) or even | (or), although at the moment match and dependent types do not allow it. It would also be useful to have predefined character classes, such as \d for a digit, and more advanced ones, such as [a-zA-Z]. Quantifiers of the form X{n} (X exactly n times) would be also relatively easy to implement. A major addition would be allowing nested capturing groups, but they would be more complicated to provide without significant changes to the algorithm.

References

[1] T. Altenkirch, C. Mcbride, and J. McKinna. Why dependent types matter. *ACM SIGPLAN Notices*, 41:1, 01 2006.

[2] O. Blanvillain, G. Schmid, M. Odersky, and V. Kuncak. Dependent types for humans. Under review, 2019.

[3] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[4] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[5] R. Chugh, D. Herman, and R. Jhala. Dependent types for javascript. *SIGPLAN Not.*, 47(10):587–606, Oct. 2012.

[6] R. A. Eisenberg. *Dependent types in Haskell: Theory and practice*. PhD thesis, University of Pennsylvania, 2016.

[7] HaskellWiki contributors. Haskellwiki: Type families. `https://wiki.haskell.org/GHC/Type_families`. Accessed on 2019-05-12.

[8] C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, Jan. 2004.

[9] Microsoft Corporation. Typescript conditional types documentation. `https://www.typescriptlang.org/docs/handbook/advanced-types.html`. Accessed on 2019-05-12.

[10] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[11] M. Odersky. Match types documentation. `https://dotty.epfl.ch/docs/reference/new-types/match-types.html`. Accessed on 2019-05-11.

[12] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In J.-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 437–450, Boston, MA, 2004. Springer US.

[13] S. Wierich. Dependent types in haskell. `https://www.youtube.com/watch?v=wNa3MMbhwS4&t=25s`. Accessed on 2019-05-12.