

Use Union Types Instead Of Either[E, A] For A More Efficient Validation?

The reason I put together this article is because I couldn't help to notice how similar Either[E, A] is to union type E | A.

And then I asked myself, could we use union types instead of Either[E, A] with a goal of getting a leaner and faster equivalent functionality?

Enough of clickbait, the answer is yes, but not by a lot, **we are talking about 4% improvement in runtime and memory usage**. Now you can chose to continue reading this blog with the right expectations.

Let's revisit how we would do validation using Either[E, A] and for comprehension. Here is a piece of code from [this blog's repo](#)

```
val employeeGood = for
  email <- UsingEither.EmailBuilder.fromString("x@dd.com")
  ssn <- UsingEither.SsnBuilder.fromString("111-11-1111")
yield Employee(email=email, ssn=ssn)
println(s"employeeGood $employeeGood")
```

I would like to clarify that in the example above we are generating Either[E, A] objects that are only used for validation and eventually thrown away once their value is extracted.

```

val employeeGood = for
  // The Either[E, A] object from UsingUnionType.EmailBuilder.fromString
  // is thrown away immediately
  email <- UsingEither.EmailBuilder.fromString("x@dd.com")
  // The Either[E, A] object from UsingUnionType.SsnBuilder.fromString
  // is also thrown away immediately
  ssn <- UsingEither.SsnBuilder.fromString("111-11-1111")
yield Employee(email=email, ssn=ssn)
println(s"employeeGood $employeeGood")

```

If we do this enough time in our code, we maybe be giving our garbage collector a lot of work just to cleanup these intermediate `Either[E, A]` wrappers!

Thus, here is the question I will try to answer in this blog:

Can we use for comprehensions for validating data without using a wrapper object like `Either[E, A]`?

I am thinking that the best candidate for this solution would be the [union types introduced in scala 3](#):

```

object UsingUnionType: {}
{}type unionWithErrorList[A] = List[Throwable] | A

object SsnBuilder:
  def fromString(string: String): unionWithErrorList[SSN] = {
    string match
      case null =>
        List(Throwable("Social security is null"))
      case _ =>
        val split = string.split("-")
        if (split.size != 3)
          List(Throwable(s"Three different sets of digits expected but ${split.size}
found"))
        else if (split(0).filter(_._isDigit)._isEmpty)
          List(Throwable(s"No digits found in area position '${string}'"))
        else if (split(1).filter(_._isDigit)._isEmpty)
          List(Throwable(s"No digits found in group position '${string}'"))
        else if (split(2).filter(_._isDigit)._isEmpty)
          List(Throwable(s"No digits found in serial position '${string}'"))
        else if (split(0).filter(!_._isDigit)._nonEmpty)

```

```
List(Throwable(s"Invalid digit found in area position '${string}'"))
    else if (split(1).filter(!_.isDigit).nonEmpty)
        List(Throwable(s"Invalid digit found in group position
        '${string}'"))
    else if (split(2).filter(!_.isDigit).nonEmpty)
        List(Throwable(s"Invalid digit found in serial position
        '${string}'"))
    else
        SSN(area = split(0).toInt, group = split(1).toInt, serial =
        split(2).toInt)
}

object EmailBuilder:
    def fromString(string: String): unionWithErrorList[Email] = string match
        case null =>
            List(Throwable("Email is null"))
        case _ =>
            val split = string.split("@")
            if (split.size != 2)
                List(Throwable(s"Email '${string}' is malformed"))
            else
                Email(user = split(0), domain = split(1))
```

But will the following work?

```
val employeeGood = for
    email <- UsingUnionType.EmailBuilder.fromString("x@dd.com")
    ssn <- UsingUnionType.SsnBuilder.fromString("111-11-1111")
yield Employee(email=email, ssn=ssn)
println(s"employeeGood $employeeGood")
```

Nope! You will get compile errors! These Union types don't have their own flatmap!

value flatMap is not a member of datavalidation.UnionTypeVsEither.UsingUnionType.unionWithErrorList[datavalidation.UnionTypeVsEither.Email], but could be made available as an extension method.

Well, then let's add a flatmap and a map to the union type to make for comprehensions work!

```
extension[B] (or: UsingUnionType.unionWithErrorList[B])
```

```
def flatMap[B1](f: B => List[Throwable] | B1): UsingUnionType.unionWithErrorList[B1] = or
match
  [case e: List[Throwable] => e
  [case o: B => f(o)

def map[B1](f: B => B1): UsingUnionType.unionWithErrorList[B1] = or match
  [case e: List[Throwable] => or.asInstanceOf[UsingUnionType.unionWithErrorList[B1]]
  case o: B => f(o).asInstanceOf[UsingUnionType.unionWithErrorList[B1]]
```

It works!

```
val employeeGood = for
  email <- UsingUnionType.EmailBuilder.fromString("x@dd.com")
  ssn <- UsingUnionType.SsnBuilder.fromString("111-11-1111")
yield Employee(email=email, ssn=ssn)
println(s"employeeGood $employeeGood")
// prints employeeGood Employee(Email(x, dd.com), SSN(111, 11, 1111))
```

But does it consume less memory? Yes, a little bit

I setup 2 benchmarks and setup the following jvm settings to collect memory usage. Make sure you are using java 11 or newer and java 8 will not recognize

"-XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC"

```
-XX: +UnlockExperimentalVMOptions -XX: +UseEpsilonGC -Xmx32g
```

The idea is that we accumulate memory usage and we track memory usage through each incremental step. Be aware this benchmark eats up a lot of memory **since we are turning off the garbage collector.**

More details about this benchmark code can be seen at the [repo file for this blog](#). Feel free to download them and run them.

```
object BenchmarkEither extends App:
  UnionTypeVsEither.employeeGenerateValidateWithEither(4_000_000)

  val t1 = System.currentTimeMillis()
  val m1 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

  val list = UnionTypeVsEither.employeeGenerateValidateWithEither(4_000_000)
  val t2 = System.currentTimeMillis()
  val m2 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()
```

```

println(s"employeeGenerateValidateWithEither: mem: gb: ${(m2 - m1) / 1_000_000_000.0}")
println(s"employeeGenerateValidateWithEither: ms: ${(t2 - t1)}")

object BenchmarkUnionType extends App:
  UnionTypeVsEither.employeeGenerateValidateWithUnionTypes(4_000_000)

  val t1 = System.currentTimeMillis()
  val m1 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

  val list = UnionTypeVsEither.employeeGenerateValidateWithUnionTypes(4_000_000)
  val t2 = System.currentTimeMillis()
  val m2 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

  println(s"employeeGenerateValidateWithUnionTypes: mem: gb: ${(m2-m1)/1_000_000_000.0}")
  println(s"employeeGenerateValidateWithUnionTypes: ms: ${(t2 - t1)}")

object NoWrappers extends App:
  UnionTypeVsEither.employeeGenerateNoValidationWrappers(4_000_000)

  val t1 = System.currentTimeMillis()
  val m1 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

  val list = UnionTypeVsEither.employeeGenerateNoValidationWrappers(4_000_000)
  val t2 = System.currentTimeMillis()
  val m2 = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()

  println(f"employeeGenerateNoValidationWrappers: mem: gb: ${(m2 - m1) /
1_000_000_000.0}% 2f")
  println(s"employeeGenerateNoValidationWrappers: ms: ${(t2 - t1)}")

```

Both benchmarks, the one for union types and the one for Either[E, A] yielded the following results rather consistently:

```

union types
  employeeGenerateValidateWithUnionTypes: mem: gb: 4.148166656
  employeeGenerateValidateWithUnionTypes: ms: 2545

either
  employeeGenerateValidateWithEither: mem: gb: 4.273995776
  employeeGenerateValidateWithEither: ms: 2639

```

One can speculate the difference should deepen the longer the for-comprehension gets (meaning more Either types being garbage collected).

Will memory efficiency help if we use opaque union types? Haven't seen any improvements

Let's try something else, **opaque types for union types**. Scala 3 introduced opaque types. The claim of opaque types is that they "provide type abstraction without any **overhead**". In Scala 2, a similar result could be achieved with [value classes](https://docs.scala-lang.org/overviews/implicit-parameters/)." (from [docs.scala-lang.org](https://docs.scala-lang.org/overviews/implicit-parameters/))

Does that mean that we could use opaque types for union types? Let's try it!

```
object UsingOpaqueUnionType:
  opaque type OpaqueUnionWithErrorList[A] = List[Throwable] | A

  object SsnBuilder:
    def fromString(string: String): OpaqueUnionWithErrorList[SSN] = {
      string match
        case null =>
          List(Throwable("Social security is null"))
        case _ =>
          val split = string.split("-")
          if (split.size != 3)
            List(Throwable(s"Three different sets of digits expected but ${split.size}
found"))
          .....
          else if (split(2).filter(!_.isDigit).nonEmpty)
            List(Throwable(s"Invalid digit found in serial position
'${string}'"))
          else
            SSN(area = split(0).toInt, group = split(1).toInt, serial =
split(2).toInt)
    }

  object EmailBuilder:
    def fromString(string: String): OpaqueUnionWithErrorList[Email] = string match
      case null =>
        List(Throwable("Email is null"))
      case _ =>
        val split = string.split("@")
        if (split.size != 2)
```

```

        List(Throwable(s"Email '${string}' is malformed"))
    else
        Email(user = split(0), domain = split(1))

extension[ B ] (or: OpaqueUnionWithErrorList[ B ])
    def flatMap[ B1 ](f: B => List[ Throwable ] | B1): OpaqueUnionWithErrorList[ B1 ] = or
match
    case e: List[ Throwable ] => e.asInstanceOf[ OpaqueUnionWithErrorList[ B1 ] ]
    case o: B => f(o).asInstanceOf[ OpaqueUnionWithErrorList[ B1 ] ]

    def map[ B1 ](f: B => B1): OpaqueUnionWithErrorList[ B1 ] =
    or match
        case e: List[ Throwable ] => or.asInstanceOf[ OpaqueUnionWithErrorList[ B1 ] ]
        case o: B => f(o).asInstanceOf[ OpaqueUnionWithErrorList[ B1 ] ]

```

Be warned your IDE may not like parsing the code above, you will likely see strange errors reporting as we are not writing standard scala.

After benchmarking opaque union types vs Either vs just union types we get the following results more or less consistently:

```

union types
employeeGenerateValidateWithUnionTypes: mem: gb: 4.148166656
employeeGenerateValidateWithUnionTypes: ms: 2545

either
employeeGenerateValidateWithEither: mem: gb: 4.273995776
employeeGenerateValidateWithEither: ms: 2639

opaque union type
employeeGenerateValidateWithOpaqueUnionTypes: mem: gb: 4.15
employeeGenerateValidateWithOpaqueUnionTypes: ms: 2493

```

The conclusion so far:

- I observed an improvement of about 4% on performance and memory savings by using union types or opaque types instead of Either[E, A]
- If we do the math on object creation (Either objects that get trashed right away) the math doesn't add up. There are clearly scala specific memory optimizations (outside the garbage collector) that make the use of Either[E, A] very efficient.

Is it justifiable to use union types instead of Either?

The answer would be: **not today.**

- Clearly, this is not a commonly accepted way of evaluating data today
- There maybe unexpected side effects when writing code like this
- IntelliJ IDE doesn't like it. You will see lots of red in the code
- I can't think of many places where it's worth the risk using Union types for a 4% improvement in performance and memory efficiency.

[Access to repo for this blog](#)

Revision #12

Created Sun, Aug 27, 2023 8:03 PM by [hernan saab](#)

Updated Sun, Oct 22, 2023 6:56 PM by [hernan saab](#)