Quantifying and Explaining Immutability in Scala

Philipp Haller
KTH Royal Institute of Technology
Stockholm, Sweden
phaller@kth.se

Ludvig Axelsson
KTH Royal Institute of Technology
Stockholm, Sweden
ludvigax@kth.se

Functional programming typically emphasizes programming with first-class functions and immutable data. Immutable data types enable fault tolerance in distributed systems, and ensure process isolation in message-passing concurrency, among other applications. However, beyond the distinction between reassignable and non-reassignable fields, Scala's type system does not have a built-in notion of immutability for type definitions. As a result, immutability is "by-convention" in Scala, and statistics about the use of immutability in real-world Scala code are non-existent.

This paper reports on the results of an empirical study on the use of immutability in several medium-to-large Scala open-source code bases, including Scala's standard library and the Akka actor framework. The study investigates both shallow and deep immutability, two widely-used forms of immutability in Scala. Perhaps most interestingly, for type definitions determined to be mutable, explanations are provided for why neither the shallow nor the deep immutability property holds; in turn, these explanations are aggregated into statistics in order to determine the most common reasons for why type definitions are mutable rather than immutable.

1 Introduction

Immutability is an important property of data types, especially in the context of concurrent and distributed programming. For example, objects of immutable type may be safely shared by concurrent processes without the possibility of data races. In message-passing concurrency, sending immutable messages helps ensure process isolation. Finally, in distributed systems immutability enables efficient techniques for providing fault tolerance.

Scala's type system does not have a built-in notion of immutability for type definitions. Instead, immutability is "by-convention" in Scala. In addition, statistics about the use of immutability in real-world Scala code are non-existent. This is problematic, since such statistics could inform extensions of Scala's type system for enforcing immutability properties.

Contributions This paper presents the first empirical results evaluating the prevalence of immutability in medium-to-large open-source Scala code bases, including the Scala standard library and the Akka actor framework [7]. We considered three different immutability properties, all of which occur frequently in all our case studies. In addition, we provide empirical results, evaluating causes for mutability of type definitions.

2 Immutability Analysis

This paper uses a notion of immutability that applies to *type definitions* rather than object references as in other work [11, 3]. For example, the definition of an immutable class implies that all its instances

V.T. Vasconcelos and P. Haller (Eds.): Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES'17)
EPTCS 246, 2017, pp. 21–27, doi:10.4204/EPTCS.246.5

are immutable. We refer to class, trait, and object definitions collectively as *templates*, following the terminology of the Scala language specification [9].

We distinguish three different immutability properties: (a) deep immutability, (b) shallow immutability, and (c) conditional deep immutability. Deep immutability is the strongest property; it requires that none of the declared or inherited fields is reassignable, and that the types of all declared or inherited fields are deeply immutable. Shallow immutability requires that none of the parents is mutable and that none of the declared or inherited fields is reassignable. Conditional deep immutability requires that none of the declared or inherited fields is reassignable, and that the types of all declared or inherited fields are deeply immutable, unless they are abstract types. For example, the type parameter `T` of the generic class `Option[T]` is abstract; type `T` is unknown within the definition of type `Option[T]`. Similarly, a Scala abstract type member [1] is treated as an abstract type. Finally, a class that declares or inherits a reassignable field (a Scala `var`) is *mutable*.

2.1 Implementation

We implement our analysis as a compiler plugin for Scala 2.11.x.[1] The plugin can be enabled when building Scala projects using the sbt or Maven build tools. The immutability analysis is implemented using Reactive Async [4] which extends LVars [6], lattice-based variables, with cyclic dependency resolution. For each template definition we maintain a "cell" that keeps track of the immutability property of the template. The value of the cell is taken from an immutability lattice; the analysis may update cell values monotonically according to the immutability lattice, based on evidence found during the analysis. For example, the cell value of a subclass is updated to `Mutable` when the analysis detects that one of the superclasses is mutable. Initially, all templates are assumed to be deeply immutable; this assumption is then updated incrementally based on evidence found by the analysis.

3 Empirical Study

We evaluate the prevalence of the immutability properties defined in Section 2 in four medium-to-large Scala open-source projects: Scala's standard library (version 2.11.8), Akka's actor package (version 2.4.17), ScalaTest (version 3.0.1), and Signal/Collect (version 8.0.2).

The Scala standard library consists of 33107 source lines of code (excluding blank lines and comments).[2] The library includes an extensive collection package [8] with both mutable and immutable collection types, as well as math, I/O, and concurrency packages such as futures [5]. Certain packages are designed to only define immutable types, including package `scala.collection.immutable` and package `scala.collection.parallel.immutable`. Other packages are designed to define mutable types, including packages `scala.collection.mutable`, `scala.collection.concurrent`, and `scala.collection.parallel.mutable`.

Akka's actor package is the standard actor implementation for Scala. ScalaTest [2] is the most widely-used testing framework for Scala. Signal/Collect [10] is a distributed graph processing framework based on Akka.

Our empirical study aims to answer the following two main research questions:

RQ1 How frequent is each immutability property for classes, traits, and objects?

[1] See `https://github.com/luax/scala-immutability-plugin`

[2] Measured using cloc v1.70, see `https://github.com/AlDanial/cloc`

Template	Occurrences	Mutable	Shallow	Deep	Cond. Deep
Class	626 (33,5%)	330 (52,7%)	54 (8,6%)	124 (19,8%)	118 (18,8%)
Case class	75 (4,0%)	19 (25,3%)	7 (9,3%)	9 (12,0%)	40 (53,3%)
Anon. class	330 (17,7%)	209 (63,3%)	26 (7,9%)	95 (28,8%)	0 (0%)
Trait	466 (25,0%)	224 (48,1%)	15 (3,2%)	93 (20,0%)	134 (28,8%)
Object	358 (19,2%)	106 (29,6%)	29 (8,1%)	223 (62,3%)	0 (0%)
Case object	12 (0,6%)	3 (25,0%)	0 (0%)	9 (75,0%)	0 (0%)
Total	**1867 (100,0%)**	**891 (47,7%)**	**131 (7,0%)**	**553 (29,6%)**	**292 (15,6%)**

Table 1: Immutability statistics for Scala standard library.

Template	Occurrences	Mutable	Shallow	Deep	Cond. Deep
Class	299 (26,8%)	115 (38,5%)	93 (31,1%)	82 (27,4%)	9 (3,0%)
Case class	206 (18,4%)	23 (11,2%)	64 (31,1%)	90 (43,7%)	29 (14,1%)
Anon. class	77 (6,9%)	33 (42,9%)	8 (10,4%)	36 (46,8%)	0 (0%)
Trait	239 (21,4%)	22 (9,2%)	17 (7,1%)	140 (58,6%)	60 (25,1%)
Object	220 (19,7%)	9 (4,1%)	47 (21,4%)	164 (74,5%)	0 (0%)
Case object	76 (6,8%)	2 (2,6%)	0 (0%)	74 (97,4%)	0 (0%)
Total	**1117 (100,0%)**	**204 (18,3%)**	**229 (20,5%)**	**586 (52,5%)**	**98 (8,8%)**

Table 2: Immutability statistics for Akka (`akka-actor` package).

RQ2 For classes/traits/objects that are not deeply immutable: what are the most common reasons why stronger immutability properties are not satisfied?

3.1 Research Question 1

Tables 1 shows the immutability statistics for Scala's standard library. One of the most important results is that *the majority of classes/traits/objects in Scala's standard library satisfy one of the immutability properties.* This confirms the intuition that functional programming with immutable types is an important programming style in Scala. Interestingly, the most common immutability property for case classes and traits is conditional deep immutability. Thus, *whether a case class or trait is deeply immutable in most cases depends on the instantiation of type parameters or abstract types.* In contrast, the majority of classes that are not case classes is mutable. Note that objects and anonymous classes cannot be conditionally deeply immutable, since these templates cannot have type parameters or abstract type members.

Table 2 shows the immutability statistics for Akka. The percentage of mutable classes/traits/objects is significantly lower compared to Scala's standard library (18.3% for Akka versus 47.7% for the Scala library).

Table 4 shows the immutability statistics for Signal/Collect. Unique to Signal/Collect is the high percentage of mutable singleton objects (46.3%), which ranges between 4.1% (Akka) and 29.6% (Scala library). However, also in Signal/Collect is the percentage of mutable case classes low compared to other kinds of templates.

Summary In our case studies, the majority of classes/traits/objects satisfy one of our immutability properties. The prevalence of mutability is especially low for case classes (with structural equality)

Template	Occurrences	Mutable	Shallow	Deep	Cond. Deep
Class	791 (36,1%)	216 (27,3%)	249 (31,5%)	288 (36,4%)	38 (4,8%)
Case class	153 (7,0%)	15 (9,8%)	81 (52,9%)	54 (35,3%)	3 (2,0%)
Anon. class	688 (31,4%)	200 (29,1%)	293 (42,6%)	195 (28,3%)	0 (0%)
Trait	227 (10,3%)	61 (26,9%)	45 (19,8%)	91 (40,1%)	30 (13,2%)
Object	254 (11,6%)	19 (7,5%)	18 (7,1%)	217 (85,4%)	0 (0%)
Case object	81 (3,7%)	2 (2,5%)	0 (0%)	79 (97,5%)	0 (0%)
Total	**2194 (100,0%)**	**513 (23,4%)**	**686 (31,3%)**	**924 (42,1%)**	**71 (3,2%)**

Table 3: Immutability statistics for ScalaTest.

Template	Occurrences	Mutable	Shallow	Deep	Cond. Deep
Class	160 (58,0%)	78 (48,8%)	24 (15,0%)	14 (8,8%)	44 (27,5%)
Case class	42 (15,2%)	4 (9,5%)	11 (26,2%)	15 (35,7%)	12 (28,6%)
Anon. class	4 (1,4%)	4 (100,0%)	0 (0%)	0 (0%)	0 (0%)
Trait	24 (8,7%)	6 (25,0%)	1 (4,2%)	3 (12,5%)	14 (58,3%)
Object	41 (14,9%)	19 (46,3%)	5 (12,2%)	17 (41,5%)	0 (0%)
Case object	5 (1,8%)	0 (0%)	0 (0%)	5 (100,0%)	0 (0%)
Total	**276 (100,0%)**	**111 (40,2%)**	**41 (14,9%)**	**54 (19,6%)**	**70 (25,4%)**

Table 4: Immutability statistics for Signal/Collect.

Reason	Immutability Property	Attribute Key
Parent type mutable (assumption)	Mutable	A
Parent type mutable	Mutable	B
Reassignable field (public)	Mutable	C
Reassignable field (private)	Mutable	D
Parent type unknown	Mutable	E
Parent type shallow immutable	Shallow immutable	F
`val` field with unknown type	Shallow immutable	G
`val` field with mutable type	Shallow immutable	H
`val` field with mutable type (assumption)	Shallow immutable	I

Table 5: Template attributes and their influence on immutability properties.

and singleton objects. Except for Signal/Collect, which is unique in this case, the majority of singleton objects are deeply immutable, ranging between 62.3% and 85.4% in our case studies. The percentage of deeply immutable *case objects* is even higher, ranging between 75% and 100%, including Signal/Collect.

In order to answer RQ2, we identified nine template *attributes*, shown in Table 5, which explain why certain immutability properties cannot be satisfied. The presence of the first five attributes forces the corresponding template to be classified as mutable. For example, a template is classified as mutable if it declares a reassignable field (attributes C and D). The last four attributes prevent the corresponding template from satisfying either deep of conditionally deep immutability. For example, if a parent class or trait is only shallow immutable (but not deeply immutable), then the corresponding template cannot be deeply immutable or conditionally deeply immutable either (attribute F).

Attribute(s)	Occurrences
B	609 (68,4%)
B C	71 (8,0%)
B C D	1 (0,1%)
B D	19 (2,1%)
B E	7 (0,8%)
C	26 (2,9%)
C D	1 (0,1%)
D	87 (9,8%)
D E	4 (0,4%)
E	66 (7,4%)

Table 6: Scala library: attributes causing mutability.

Attribute(s)	Occurrences
F	28 (21,4%)
F G	5 (3,8%)
F G H	1 (0,8%)
F H	4 (3,1%)
F J	6 (4,6%)
G	22 (16,8%)
G H	4 (3,1%)
G H J	3 (2,3%)
G J	2 (1,5%)
H	40 (30,5%)
H J	3 (2,3%)
J	7 (5,3%)

Table 7: Scala library: attributes causing shallow immutability (instead of deep immutability).

3.2 Research Question 2

Tables 6 and 7 show the causes for mutability and shallow immutability, respectively, for the Scala library. The main cause for a template to be classified as mutable is the existence of a parent which is mutable. Important causes for templates to be classified as shallow immutable rather than deeply immutable are (a) the existence of a non-reassignable field with a mutable type (attribute H), and (b) the existence of a parent which is shallow immutable (attribute F).

Tables 8 and 9 show the causes for mutability and shallow immutability, respectively, for Akka actors. The main cause for a template to be classified as mutable is the existence of a parent which is mutable; this matches the statistics of the Scala library. Other important causes are (a) parent types whose immutability is unknown (*e.g.,* due to third-party libraries for which no analysis results are available) and (b) private reassignable fields. Unlike the Scala library, the most important cause for shallow immutability (rather than deep immutability) in Akka are non-reassignable fields of a type whose immutability is unknown; this suggests that the absence of analysis results for third-party libraries has a significant impact on the classification of a type as shallow immutable rather than deeply immutable. On the other hand, this means that the actual percentage of deeply immutable templates may be even higher. Therefore, an important avenue for future work is to enable the analysis of third-party libraries. The second most important cause is the existence of a parent which is shallow immutable (attribute F).

4 Conclusion

Immutability is an important property of data types, especially in the context of concurrent and distributed programming. For example, objects of immutable type may be safely shared by concurrent processes without the possibility of data races. In message-passing concurrency, sending immutable messages helps ensure process isolation. In this paper we presented the first empirical results evaluating the prevalence of immutability in medium-to-large open-source Scala code bases, including the Scala standard library and the Akka actor framework. We considered three different immutability properties,

Attribute(s)	Occurrences
A	3 (1,5%)
A B D	1 (0,5%)
A E	1 (0,5%)
B	76 (37,3%)
B C	3 (1,5%)
B C D	1 (0,5%)
B D	6 (2,9%)
B E	6 (2,9%)
C	7 (3,4%)
C D	2 (1,0%)
C D E	1 (0,5%)
D	24 (11,8%)
D E	1 (0,5%)
E	72 (35,3%)

Table 8: Akka: attributes causing mutability.

Attribute(s)	Occurrences
F	38 (16,6%)
F G	9 (3,9%)
F G H	2 (0,9%)
F G J	3 (1,3%)
F H	3 (1,3%)
F J	3 (1,3%)
G	94 (41,0%)
G H	8 (3,5%)
G H I	1 (0,4%)
G H J	1 (0,4%)
G J	16 (7,0%)
H	22 (9,6%)
H J	4 (1,7%)
J	25 (10,9%)

Table 9: Akka: attributes causing shallow immutability (instead of deep immutability).

all of which occur frequently in all our case studies. In our case studies, the majority of classes/traits/objects satisfy one of our immutability properties. The prevalence of mutability is especially low for case classes (classes with structural equality) and singleton objects. The most important causes for mutability are mutable parent classes and private reassignable fields. To our knowledge we presented the first empirical study of its kind. We believe our insights are valuable both for informing the further evolution of the Scala language, and for designers of new wide-spectrum languages, combining functional and imperative features.

References

[1] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf & Sandro Stucki (2016): *The Essence of Dependent Object Types*. In: *A List of Successes That Can Change the World*, Springer, pp. 249–272, doi:10.1007/978-3-319-30936-1_14.

[2] Artima, Inc. (2009): *ScalaTest*. http://www.scalatest.org.

[3] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield & Joe Duffy (2012): *Uniqueness and reference immutability for safe parallelism*. In: *OOPSLA*, ACM, pp. 21–40, doi:10.1145/2384616.2384619.

[4] Philipp Haller, Simon Geries, Michael Eichberg & Guido Salvaneschi (2016): *Reactive Async: Expressive Deterministic Concurrency*. In: *ACM SIGPLAN Scala Symposium*, ACM, pp. 11–20, doi:10.1145/2998392.2998396.

[5] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn & Vojin Jovanovic (2012): *Futures and promises*. http://docs.scala-lang.org/overviews/core/futures.html.

[6] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami & Ryan R. Newton (2014): *Freeze after writing: quasi-deterministic parallel programming with LVars*. In: *POPL*, ACM, pp. 257–270, doi:10.1145/2535838.2535842.

[7] Lightbend, Inc. (2009): *Akka*. http://akka.io/.

[8] Martin Odersky & Adriaan Moors (2009): *Fighting bit Rot with Types (Experience Report: Scala Collections)*. In: *FSTTCS*, *LIPIcs* 4, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 427–451, doi:10.4230/LIPIcs.FSTTCS.2009.2338.

[9] Martin Odersky et al. (2014): *The Scala Language Specification Version 2.11*. Available at `http://www.scala-lang.org/files/archive/spec/2.11/`.

[10] Philip Stutz, Abraham Bernstein & William W. Cohen (2010): *Signal/Collect: Graph Algorithms for the (Semantic) Web*. In: *ISWC*, Springer, pp. 764–780, doi:10.1007/978-3-642-17746-0_48.

[11] Matthew S. Tschantz & Michael D. Ernst (2005): *Javari: adding reference immutability to Java*. In: *OOPSLA*, ACM, pp. 211–230, doi:10.1145/1094811.1094828.