# Adding Reference Immutability to Scala

by

Yaoyu Zhao

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

**Abstract**

Scala is a multi-paradigm programming language combining the power of functional and object-oriented programming. While Scala has many features promoting immutability, it lacks a built-in mechanism for controlling and enforcing reference immutability. Reference immutability means the state of an object and all other objects reachable from it cannot be mutated through an immutable reference. This thesis presents a system for reference immutability in Scala, along with a simple implementation in the Dotty (Scala 3) compiler. By extending the Scala type system and encoding mutability as types within annotations, my system enables tracking and enforcing reference immutability for any type. It addresses challenges such as the complexities of the Scala type system and context sensitivity with nested classes and functions. The design offers binary compatibility with existing Scala code, and promotes predictable object behavior, reducing the risk of bugs in software development.

**Acknowledgements**

I would like to thank my advisor, Ondřej Lhoták, for his invaluable guidance and unwavering support throughout my undergraduate and master's studies. His second-year compiler course inspired me to delve deeper into the fascinating world of compilers and type systems. I am truly grateful for his patience and insightful mentorship in shaping my research path.

I would like to express my appreciation to Peter Buhr and Yizhou Zhang for their contributions as members of my thesis committee. Their thoughtful comments and suggestions have greatly enriched the quality of my thesis.

A special thanks goes to Prabhakar Ragde for his exceptional teaching of courses related to functional programming and programming languages. Learning from him has been a pleasure, and his guidance provided me with an opportunity to enter the world of research in this field.

I am grateful to my fellow labmates in PLG, including Edward Lee, Ende Jin, James You, Jianlin Li, and others. Thanks for discussing with me, taking coffee walk, having dinner together, and supporting me all the time.

Finally, I am deeply appreciative of my family and friends for their support, understanding, and encouragement. Their belief in me has been a constant source of motivation and inspiration throughout my academic journey.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Software bugs caused by mutable state are a significant source of frustration and cost in modern software development. Immutability refers to the state of a variable of an object cannot be changed after it is created. It has been widely advocated by API and programming language designers as a way to mitigate bugs and enhance security, with additional benefits including simple state management, thread-safety, and safe and efficient sharing [6]. Despite the advantages of immutability, mutation is still preferred in certain contexts due to its ability to express algorithms naturally and optimize performance.

The functional programming language community is particularly concerned with immutability and purity. Pure programming paradigm emphasizes referential transparency and does not rely on side effects. Pure functions can also benefit concurrency and parallelism since they can be safely executed in any order without the risk of race conditions or deadlocks. Code written in a pure functional style is easier to test and it is easier to reason about its correctness and safety [1].

Reference immutability, which guarantees that the abstract state of an object cannot be mutated through an immutable reference, is a key aspect of immutability in object-oriented programming languages. The abstract state is transitive, including the state of the object and all other objects reachable from it. It is used to give developers more information about whether an object can be mutated at a certain point. It is possible that an object is referred to by a mutable reference and a read-only reference at the same time. This work focuses on reference immutability, which guarantees that only mutable references can modify an object, while object immutability ensures that only immutable references point to an immutable object (an object which cannot be mutated after creation). Reference immutability has been extensively studied in existing object-oriented programming languages

such as Java [19, 18, 13], and formalizations of immutability systems in Featherweight Java have been demonstrated at [15].

Scala is a multi-paradigm programming language integrating aspects of both functional and object-oriented programming. It is widely used for developing complex applications in various domains, including web services, big data processing, and scientific computing. Scala includes several features that encourage developers to write pure and functional code, such as immutable variables and collections, lazy evaluation, pattern matching, and tail recursion. These features facilitate code reuse and reduce the need for mutation. However, Scala lacks a built-in mechanism for controlling and enforcing reference immutability.

## 1.1 Previous Approaches on Reference Immutability

Javari and ReIm implemented reference immutability for Java [18, 13]. They modify the type system of Java and add new keywords to express and enforce immutability constraints. As shown in the following example, the type qualifier `readonly` denotes that `rd` is an immutable reference. Hence, any mutation operation on `rd` causes a compile-time error.

```
Date md = new Date();  // mutable Date by default
readonly Date rd = md; // an immutable reference of Date
md.setHours(1); // OK, md is mutable
rd.setHours(2); // compile time error, rd is immutable
```

Listing 1.1: An exmaple of immutability in Java.

ReIm also introduced *context sensitivity* for immutability, encoded using the concept of *viewpoint adaptation* from Universe Types [7], which enables immutability to be transitive. Simply speaking, the references of objects referred to by an object with read-only reference are also read-only. This ensures the data and behaviours related to a reference are consistent. The following example illustrates viewpoint adaptation expressed by a special mutability qualifier `polyread`. Both the receiver type and the result type of the method `getData` are qualified by `polyread`. When the method is invoked on a mutable reference `dc1`, the result type is mutable `Data`; when it is invoked on a read-only reference `dc2`, the result type is `readonly Data`.

```
abstract class DateCell {
  public polyread Data getData(polyread DateCell this);
}

DataCell dc1 = ...
Data d1 = dc1.getData()
```

```
readonly DataCell dc2 = ...
readonly Data d2 = dc2.getData()
```

Listing 1.2: An exmaple of viewpoint adaptation.

However, these approaches are limited in their usage on generic types and do not allow for further operations on the qualifiers due to constraints in the Java grammar. Additionally, nested code is not discussed in their works, which is common in practice (for example, anonymous classes and lambda expressions).

In contrast, Scala has a richer type system that enables users to benefit from versatile generic types and various type operations, such as match types, union and intersection types, and path-dependent types, allowing for more expressive code. Nested methods and classes are also common in Scala and are the building blocks of functional programming. Efforts have been made to ensure a smooth transition between Scala 2 and 3, and hence, adding a new feature should not require programmers to throw away old code. These requirements must be considered when designing a new feature for Scala.

The only existing research on implementing a reference immutability system for Scala is the work on roDOT [9], which is a complex extension of DOT calculus [4]. The work on roDOT only presents a calculus that can serve as a basis for implementation in the Scala compiler, rather than providing a complete implementation itself. Additionally, the consideration of backward compatibility is not addressed in this work.

## 1.2 Reference Immutability Issue in Scala

While most Scala code is immutable by default [12], mutable objects may sometimes be preferred for reasons such as simplicity and efficiency.

Consider the following Scala code modified from the *Dotty* compiler that defines nodes of an abstract syntax tree (AST):

```
type Untyped = Type | Null

abstract class Tree[+T <: Untyped]:
  protected var myTpe: T = uninitialized

  final def tpe: T = myTpe

  def withType(tpe: Type)(using Context): Tree[Type] =
```

```
    myTpe = tpe
    this.asInstanceOf[Tree[Type]]
```

Listing 1.3: AST from Dotty

The class `Tree` is parameterized by a type parameter `T`, which indicates whether the abstract syntax tree (AST) is typed (`T` is `Type`) or untyped (`T` is `Untyped`). The tree has a protected field `myTpe` with type `T` to store the associated type. If it is an untyped tree, it is expected to not get a type from the tree. The `withType` method can update the associated type in place for efficiency, and returns a typed tree that is isomorphic to this tree. This method is called only when it is safe to do so under sharing.

This implementation of `withType` is straightforward, but it has a potential issue if a tree is shared in multiple places and updated with different types. Consider the following code snippet:

```
def assignType(tree: Tree[Untyped], tp: Type): Tree[Type] =
  val newTree = tree.withType(tp)
  processTree1(newTree)
  processTree2(newTree)
  newTree
```

Listing 1.4: A dangerous example to handle AST.

Looking at this code, a programmer may expect that the tree passed to `processTree2` has the type `tp`, and the correct behaviour of the code may depend on this expectation. However, this expectation may be violated if `processTree1` unexpectedly calls `withType` to change the type of the tree.

Suppose these is a type annotation `@readonly` can be used to indicate that a reference is immutable. Functions `processTree1` and `processTree2` can be redefined as following:

```
def processTree1(tree: Tree[Type] @readonly): Unit = ...
def processTree2(tree: Tree[Type] @readonly): Unit = ...
```

Listing 1.5: Potential fix.

This way, developers know that the tree should remain immutable at these places, and any modification causes a warning or error during compilation. Additionally, the type of `val newTree` can be declared as `@readonly` to ensure that none of the methods that it is passed to can mutate it.

Enforcing reference immutability through a robust type system ensures predictable object behavior and prevents unintended mutations. This approach in turn can help reduce the risk of bugs and make code easier to reason about.

## 1.3 Challenges of Implementing in Scala

Designing and implementing reference immutability in Scala presents unique challenges that are not present in Java.

One significant challenge is the complexity of the Scala type system. In Java, a type is a primitive type, a class, or a generic type. On the other hand, Scala's type system includes intersection and union types, singleton types, and match types. The set of values a type represents and the subtyping relationships among different types are more complicated. When introducing a general type qualifier in Scala, the compiler cannot directly control the qualified type to be a class. Therefore, incorporating a mutability constraint into a type must consider and accommodate these complexities appropriately.

Another challenge is the presence of nested classes and functions. In Scala, nested classes and functions have access to the entire state of their enclosing scope. This creates difficulties in determining the mutability of an object, particularly when it is referenced from a member function of a nested class. Resolving this challenge requires a careful consideration of the scoping rules and appropriate handling of mutability constraints within nested contexts.

Additionally, practical considerations add to the challenge of designing and implementing a new feature in Scala. One such consideration is the need to handle existing Scala projects. Any new features or changes to the compiler must be backward-compatible with existing code. Furthermore, the changes to the compiler should be as minimal as possible to minimize disruption. These practical considerations need to be balanced with the desire to provide a robust and effective feature.

## 1.4 Solution

To handle the complexities of the Scala 3 type system, I carefully design the encoding of mutability with compatibility in mind. Mutabilities are represented as types within type annotations, and mutability operations are encoded as union and intersection type operations. This encoding allows for elegant checking of mutabilities through subtyping comparison.

Another key aspect of the solution is addressing the issue of parametric mutability with nested classes and functions. To tackle this, a more general and powerful design is developed for viewpoint adaptation, based on the previous approach of `polyread`. This

design allows the compiler to handle nested code by appropriately deriving the mutability of a reference based on its enclosing scope.

The solution presented in this thesis successfully addresses the challenges of the Scala 3 type system and nested code, allowing for the enforcement of reference immutability. Moreover, it provides a foundation for supporting arbitrary constraints on types. This flexibility opens up possibilities for extending the system to accommodate other type-related constraints and enables the development of more expressive and robust programs in Scala.

## 1.5 This Thesis

This thesis presents a system of reference immutability in Scala along with a simple implementation in the *Dotty* (Scala 3) compiler. My contributions can be summarized as follows:

- An expressive and useful design of reference immutability in Scala. The extension to the type system enables tracking immutability on types without breaking the complicated type system already in place.
- Flexible mutability constraints purely from types. I encode mutability in types and mutability comparison in subtyping to support expressing flexible mutability constraints.
- A formal model to describe reference immutability in Scala. I present the subtyping and typing rules for my system as an extension to System $\mathtt{F}_{<:>}$.
- A simple implementation in the *Dotty* compiler. The implementation closely follows the rules described in the formal model. The changes to the compiler are enclosed within a module to minimize the impact on other parts of the compiler.
- Binary compatibility to existing Scala code. I have carefully designed the new types to ensure backward compatibility both before and after a project's migration to this feature.

### 1.5.1 Outline

The remaining chapters of this thesis are organized as follows. Chapter 2 provides an overview of the reference immutability design, including a formal model for the core language. It also discusses additional rules specific to Scala language. Chapter 3 focuses on describing the implementation of the reference immutability system in the *Dotty* compiler and backward compatibility. In Chapter 4, various projects and tests are used to evaluate both the design and implementation, and the results of the evaluation are presented. Chapter 5 surveys related work in the field of reference immutability. Finally, Chapter 6 concludes the thesis by summarizing the contributions and discussing future directions of research.

# Chapter 2

# Reference Immutability in Scala

The reference immutability in Scala is designed with the following constraints in mind to make it practical and easy to adopt.

- **Expressive and Simple:** The system should be expressive enough to support common use cases, yet simple enough to be adopted by users and implemented in the current compiler.
- **Gradual:** Existing Scala code should be able to interact with mutability-checked code.
- **Backwards compatible:** The system should be compatible with Scala's type system, such that code with mutability annotations can compile even with mutability checking turned off.
- **Safe and efficient:** The system should be safe and efficient without introducing extra runtime overhead in compiled code.

Section 2.1 provides a brief introduction to union and intersection types in Scala 3. Section 2.2 describes the basic definition of mutability qualifiers and mutability types. The formal model is presented in Section 2.3, including a syntax, subtyping rules, and typing rules of the core language. Furthermore, Section 2.4 covers additional rules specific to Scala. Finally, an overview of a class structure containing different types of fields is provided in Section 2.5.

## 2.1 Background of Union and Intersection Types

Before delving into the definition of mutabilities, it is important to provide a brief introduction to union and intersection types in Scala 3. They are new additions to Scala 3 that allow for the representation of combinations of types. In the subsequent sections, union and intersection types are used to represent various combinations of mutabilities within the context of reference immutability.

### 2.1.1 Union Types

The `|` operator is used on types to create a union type. The resulting type, `T1 | T2`, represents values that are either of type `T1` or of type `T2`. This operator is commutative, meaning that `T1 | T2` is the same as `T2 | T1`. According to the subtyping rules, `T1` is a subtype of `T1 | T2` and `T2` is a subtype of `T1 | T2` because a value of type `T1` or `T2` can be used wherever a value of type `T1 | T2` is required.

As shown in Listing 2.1, the `readBook` method accepts a parameter named `query` with the union type `BookName | BookID`. The argument passed to it can be either a `BookName` or a `BookID`:

```
case class BookName(name: String)
case class BookID(id: ISBN)

// Read a book according to the name or ISBN
def readBook(query: BookName | BookID) =
  val book = query match
    case BookName(name) => lookupName(name)
    case BookID(id) => lookupISBN(id)
  // handle book here
  ...
```

Listing 2.1: An example of union types.

This code is a flexible and type-safe solution that does not require the developer to create an explicit hierarchy of classes wrapping the two types of input. Attempting to pass in a value of type other than `BookName` or `BookID` results in an error:

```
readBook("Programming in Scala") // error
// Found:    ("Programming in Scala" : String)
// Required: BookName | BookID
```

Listing 2.2: The error of passing wrong type to a union type.

### 2.1.2 Intersection Types

Intersection types, denoted by `T1 & T2`, represent values that have both the type `T1` and `T2` simultaneously. The members (methods and fields) of `T1 & T2` are all members that belong to either `T1` or `T2`. The `&` operator is commutative as well, meaning that `T1 & T2` is equivalent to `T2 & T1`. According to the subtyping rules, `T1 & T2` is a subtype of `T1` and `T2`, because a value of type `T1 & T2` can be used wherever a value of type `T1` or `T2` is required.

Intersection types can be useful to describe requirements structurally. As shown in Listing 2.3, the method `printAndClose` takes a parameter of type `Printable[Book] & Closeable`. This method can accept any value as long as it is a subtype of both `Printable[Book]` and `Closeable`.

```
trait Printable[A]:
  def println(a: A): Unit

trait Closeable:
  def close(): Unit

val book: Book = ...

def printAndClose(p: Printable[Book] & Closeable) =
  p.println(book)
  p.close()
```

Listing 2.3: An example of intersection types.

The use of an intersection type eliminates the need for a nominal helper trait like `PrintableAndCloseable[A] extends Printable[A], Closeable`. Instead, a class can choose to extend the traits that it needs.

## 2.2 Basic Definitions

The type `T @mut[readonly | M]` is an example of a type with mutability.

- `@mut[...]` is a mutability annotation, which adds mutability constraints to a type. Incorporating types with mutability is disscussed in Section 2.2.3.
- `readonly | M` is a mutability. It is a combination of a mutability qualifier and a mutability variable `M`. These concepts are described in Section 2.2.2.

- `readonly` is a mutability qualifier, which is a basic component of mutability. Mutability qualifiers are introduced in Section 2.2.1.

### 2.2.1 Mutability Qualifiers

My mutability system adds three mutability qualifiers to Scala: `mutable`, `polyread` and `readonly`. These qualifiers serve as the fundamental components of mutability. They are implemented as special annotation classes in the standard library and are defined with a natural subtyping relation of `mutable <: polyread <: readonly`. This subtyping relation is established because a *mutable* reference can be used anywhere that any reference is expected, and any reference can be used where a *readonly* reference is expected.

`polyread` is a special mutability qualifier, which is introduced as a convenience in expressing polymorphic mutability on member methods without changing their signature. It represents a possible mutability between `mutable` and `readonly`. When a function is invoked and some of its parameter types contain `polyread` annotations, all occurrences of `polyread` within the function type are instantiated to a specific mutability based on the context of the function call.

The concept of `mutable`, `readonly`, and `polyread` was originally introduced by Javari and ReIm [18, 13]. In my system, these qualifiers carry similar meanings but are implemented with different rules to align with the more powerful type system and syntax of Scala.

### 2.2.2 Mutability

Mutability is a type that is a subtype of `readonly` and a supertype of `mutable`. A valid mutability can be: a mutability qualifier, a type variable bounded by mutability, or an intersection or union type with mutability on both sides.

Users can define a type parameter `[M >: mutable <: readonly]` in Scala to represent polymorphic mutability. When the type parameter `M` is instantiated with an argument, the argument must be a supertype of `mutable` and a subtype of `readonly`. When a type parameter is bounded by mutabilities, it is referred to as a *mutability variable*.

Intersection and union types can be used to represent the minimum and maximum of two mutabilities. Given two mutabilities `M1` and `M2`, where `M1 <: M2`, the maximum mutability (also referred as the highest) is the union of them: `M1 | M2`. This follows the

subtyping rule: `M1 <: (M1 | M2) =:= M2` (where `=:=` designates subtyping in both directions). Likewise, the minimum mutability (also referred as the lowest) is the intersection of them: `M1 & M2`, and `M1 =:= (M1 & M2) <: M2`. This rule is used when combining multiple mutability annotations on a single type or computing the mutability of intersection or union types. The use is presented in the next section.

The mutability qualifiers are special marker types in my system and cannot be instantiated or annotated with any mutability. When mutability qualifiers are used as value types or combined with value types, the type does not hold any special meaning, such as a function type `readonly => mutable` and a union type `readonly & String`. The two example types are not subtypes of `readonly` and supertypes of `mutable`. Consequently, the type system prohibits their usage as mutabilities.

### 2.2.3 Types with Mutability

Scala 3 includes an experimental feature that allows annotations with type parameters. Utilizing this feature, the *mutability annotation* is defined as `class mut[M >: mutable <: readonly]` with one type parameter that accepts a mutability, such as `@mut[mutable]` or `@mut[readonly | mutable]`.

The mutability qualifiers can be used as annotations directly, like `@mutable`, `@readonly`, and `@polyread`. These are desugared to `@mut[mutable]`, `@mut[readonly]`, and `@mut[polyread]`, respectively. To attach a mutability other than these qualifiers, it must be placed within the `@mut` annotation.

The goal of utilizing annotations to attach mutability to a type is to provide users with an intuition that the underlying type's meaning remains unchanged while restrictions on mutation are imposed.

- `T @mutable`: An object with a *mutable* type can be mutated. As a qualifier, `@mutable` keeps the original mutability of `T`. For compatibility with existing Scala code, this annotation is optional, and the type `T` is equivalent to `T @mutable` for any type `T`.
- `T @readonly`: `@readonly` gives the highest mutability to the type. An object with a *readonly* type, and any objects referred to by it, cannot be mutated. Specifically, assignment to a field of a `readonly` reference is prohibited (after initialization), and all references reachable from this object are also `readonly`. This restriction ensures that the object and its references remain immutable.

- `T @mut[M]`: An object with a type annotated by `@mut[M]` has mutability that depends on some mutability `M`. `M` can be a concrete mutability qualifier, a mutability variable, or a combination of mutabilities. This restriction is particularly useful when the mutability of a type is uncertain.

- `T @polyread`: The mutability of a *polyread* type is polymorphic, meaning that the reference can be either mutable or readonly, and all polyread references in the same function scope have the same mutability. The mutability of a `polyread` reference is determined by its use in the program. It has special rules and restricted usage in the system. The scope of a `polyread` is always implicitly the innermost method in which it is used. Therefore, `polyread` can be used without modifying the signature of any method or class to add a mutability type parameter. Unlike `@polyread`, `@mut[M]` can express more complex mutability bounds and dependencies beyond functions. However, the mutability variable must be explicitly declared somewhere as a type parameter of some method or class.

For each reference type `T`, `T @readonly` is a supertype of `T`. The new type hierarchy is shown in figure 2.1. The black part is the original type hierarchy, and the red part is the newly added types and subtype relations.

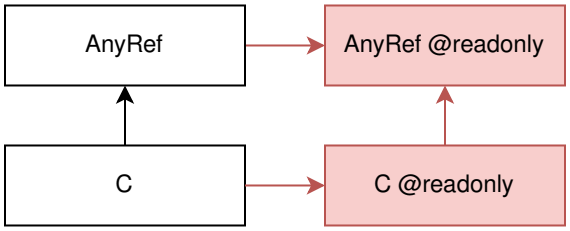


Figure 2.1: The new type hierarchy in reference immutability.

Types in this setting can have multiple mutability annotations, unlike in existing work on Java [18, 13]. This can happen if a type parameter is annotate with mutability, and it is instantiated with a type with other mutability annotations. The mutability of a type is defined as the *highest* mutability of its annotations, which can be nicely expressed as the *union* of all the mutabilities from annotations. For instance, `T @readonly @mutable` is a valid type and is equivalent to `T @readonly`, which can be shown as `T @mut[readonly] @mut[mutable] =:= T @mut[readonly | mutable] =:= T @mut[readonly]`.

The context sensitivity of reference immutability refers to the mutability of reachable objects being dependent on the mutability of the object reference itself [13], which is

usually achieved through viewpoint adaptation from Universe Types [7]. For instance, if an object reference `a` is read-only and has a mutable field `x`, the expression `a.x` is assigned a readonly type, even though the type of `x` is mutable at its definition site. To enable viewpoint adaptation, a key requirement of the implementation is that the compiler must be able to, given any type, determine the mutability of that type. The mutability of a type must be expressible and not partial. For example, given a member selection `a.b`, the mutability of this selection depends on the mutability of `a`. If the mutability of `a` cannot be expressed as a type, then its mutability cannot be passed to the type of the selection.

Therefore, the lower and upper bounds of each type parameter must have the same mutability so that that type parameter can itself be given that mutability. The function signature `[T >: C1 @mutable <: C2 @readonly](x: T): T` is not allowed, because the mutability of `T` is uncertain and cannot be expressed in the current system. Note that type `T` itself is not a mutability, and `@mut[T]` is not a valid expression. Instead, the effect can be achieved by introducing a mutability variable as follows: `[M >: mutable <: readonly, T >: C1 <: C2](x: T @mut[M]): T @mut[M]`, such that, the mutability of `T @mut[M]` is `M`. While this might appear similar to *polyread*, it is not appropriate to say the mutability of `T >: C1 @mutable <: C2 @readonly` is `polyread` because the mutability of a type parameter can be independent from function's *polyread* parameters. The definition and rules for `polyread` are given in Section 2.4.2.

While existing work deals with reference immutability in a setting with classes, Scala 3 adds intersection and union types, which need to be accounted for in the reference immutability system.

Since the mutability annotations do not affect the run-time behaviour, a *readonly* reference has the same run-time representation as the underlying reference. For the union type `C | C @readonly`, while it is sound to distinguish `C` or `C @readonly`, doing so in Scala is challenging as the objects of both are only known to be of `class C` at run time. To resolve this, intersection and union types are weakened to select the smallest (resp. highest) mutability from both sides of the operation. Thus, the mutability of the union type `T1 | T2` is the largest mutability of both sides of the union, and the mutability of an intersection type `T1 & T2` is the smallest mutability of both sides of the intersection. This avoids having a type that is partially read-only and partially mutable. For instance, `C | C @readonly` is equivalent to `C @readonly`.

This rule can be stated using intersection and union types on the mutability as well. For example, the mutability of `T1 @mut[M1] & T2 @mut[M2]` is `M1 & M2`, and the mutability of `T1 @mut[M1] | T2 @mut[M2]` is `M1 | M2`.

Furthermore, this rule helps to determine the mutability of methods. Suppose class `A`

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | $S, T, M$ | ::= | **Types** |
| $s, t$ | ::= | | **Terms** | | $\top$ | top type |
| | $\lambda(x : S).t$ | term abstraction | | $\bot$ | bottom type |
| | $\Lambda(X : S..T).t$ | type abstraction | | Mutable | mutable quantifier |
| | $x$ | term variable | | Readonly | readonly quantifier |
| | $s(t)$ | application | | $X$ | type variable |
| | $s[T]$ | type application | | $S \to T$ | function type |
| | $\{x_1 : s_1, x_2 : s_2, \ldots\}$ | records | | $\forall(X : S_1..S_2).T$ | abstraction type |
| | $s.x$ | field read | | $\{x : T\}$ | record type |
| | $s.x = t$ | field write | | $S \mid T$ | union type |
| | | | | $S \,\&\, T$ | intersection type |
| | | | | $T$ `@mut[M]` | mutability type |

Figure 2.2: The core syntax of reference mutability.

has two subclasses, `B` and `C`, and `A` has a method `f` that mutates a field. `f` should only be invoked on a *mutable* reference. If an object has type `B @readonly | C`, it becomes challenging for the compiler to decide whether `f` is callable from this type, as it has to compute callable members using mutability from both sides and make an intersection. By applying the above rule, the mutability of `B @readonly | C` can be easily determined to be `readonly`, making it forbidden to call `f` on this object.

## 2.3 Formal Model

### 2.3.1 Core Language

Figure 2.2 shows the syntax of the core language of reference immutability, based on System $\mathtt{F}_{<:>}$ [2] with records, intersection, and union types; changes from System $\mathtt{F}_{<:>}$ are noted in grey.

The mutability qualifiers `Mutable` and `Readonly` are defined as first-class types. Since `polyread` can be desugared into type parameters, it is not included in the core language.

### 2.3.2 Subtyping Rules

The mutability of a type `T` as `Mut(T)` is formally defined in Figure 2.3. The definition follows the rules described in Section 2.2.3.

$$
\begin{aligned}
\texttt{Mut}(X) &= \texttt{Mut}(T) \text{ (where } X \texttt{ >: } S \texttt{ <: } T \in \Gamma) \\
\texttt{Mut}(S \mid T) &= \texttt{Mut}(S) \mid \texttt{Mut}(T) \\
\texttt{Mut}(S \;\&\; T) &= \texttt{Mut}(S) \;\&\; \texttt{Mut}(T) \\
\texttt{Mut}(T \texttt{ @mut[M]}) &= \texttt{Mut}(T) \mid M \\
\texttt{Mut}(\_) &= \texttt{Mutable}
\end{aligned}
$$

Figure 2.3: Definition of the mutability of a type `T`.

As an example, the process of computing the mutability of `(A @readonly & B) | C @readonly` is shown in Figure 2.4, where `A`, `B`, and `C` are regular classes.

$$
\begin{aligned}
&\texttt{Mut}((A \texttt{ @mut[Readonly]} \;\&\; B) \mid C \texttt{ @mut[Readonly]}) \\
&= \texttt{Mut}(A \texttt{ @mut[Readonly]} \;\&\; B) \mid \texttt{Mut}(C \texttt{ @mut[Readonly]}) \\
&= (\texttt{Mut}(A \texttt{ @mut[Readonly]}) \;\&\; \texttt{Mut}(B)) \mid \texttt{Mut}(C \texttt{ @mut[Readonly]}) \\
&= (\texttt{Readonly} \;\&\; \texttt{Mutable}) \mid \texttt{Readonly} \\
&= \texttt{Mutable} \mid \texttt{Readonly} \\
&= \texttt{Readonly}
\end{aligned}
$$

Figure 2.4: Computing the mutability of `(A @readonly & B) | C @readonly`.

In this way, any type can be split into two parts: the underlying type and the mutability. Since mutabilities are normal types, checking subtyping of types with mutabilities becomes checking subtyping of the mutabilities and checking subtyping of the underlying types.

Figure 2.5 shows the subtyping rules ( `<:` ) of the core system, which is presented as an extension to the original subtyping relation. `<:` is the subtyping relation with mutability

**Subtyping** $\boxed{\Gamma \vdash S <: T}$

$$\Gamma \vdash T <:' T \quad \text{(REFL)}$$

$$\Gamma \vdash S <:' \top \quad \text{(TOP)}$$

$$\Gamma \vdash \bot <:' S \quad \text{(BOTTOM)}$$

$$\Gamma \vdash \texttt{Mutable} <:' \texttt{Readonly} \quad \text{(MUTABLE-READONLY)}$$

$$\frac{\Gamma \vdash R <:' S \qquad \Gamma \vdash S <:' T}{\Gamma \vdash R <:' T} \quad \text{(TRANS)}$$

$$\frac{X : S..T \in \Gamma}{\Gamma \vdash X <:' T} \quad \text{(TVAR-UPPER)}$$

$$\frac{X : S..T \in \Gamma}{\Gamma \vdash S <:' X} \quad \text{(TVAR-LOWER)}$$

$$\frac{\Gamma \vdash T_1 <:' S_1 \qquad \Gamma \vdash S_2 <:' T_2}{\Gamma \vdash S_1 \to S_2 <:' T_1 \to T_2} \quad \text{(ARROW)}$$

$$\frac{\begin{array}{c}\Gamma \vdash R_1 <:' R_2, S_2 <:' S_1 \\ \Gamma, X : R_2..S_2 \vdash T_1 <:' T_2\end{array}}{\Gamma \vdash \forall(X : R_1..S_1).T_1 <:' \forall(X : R_2..S_2).T_2} \quad \text{(ALL)}$$

$$\frac{\Gamma \vdash S <:' T \qquad \Gamma \vdash T <:' S}{\Gamma \vdash \{x : S\} <:' \{x : T\}} \quad \text{(RECORD)}$$

$$\Gamma \vdash S \,\&\, T <:' S \quad \text{(INTER-LEFT)}$$

$$\Gamma \vdash S \,\&\, T <:' T \quad \text{(INTER-RIGHT)}$$

$$\frac{\Gamma \vdash S <:' T_1 \qquad \Gamma \vdash S <:' T_2}{\Gamma \vdash S <:' T_1 \,\&\, T_2} \quad \text{(INTER)}$$

$$\Gamma \vdash S <:' S \mid T \quad \text{(UNION-LEFT)}$$

$$\Gamma \vdash T <:' S \mid T \quad \text{(UNION-RIGHT)}$$

$$\frac{\Gamma \vdash S_1 <:' T \qquad \Gamma \vdash S_2 <:' T}{\Gamma \vdash S_1 \mid S_2 <:' T} \quad \text{(UNION)}$$

$$\frac{\Gamma \vdash S <:' T}{\Gamma \vdash S \ \texttt{@mut[M]} <:' T} \quad \text{(M-IGNORE-1)}$$

$$\frac{\Gamma \vdash S <:' T}{\Gamma \vdash S <:' T \ \texttt{@mut[M]}} \quad \text{(M-IGNORE-2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{wf}\ S \qquad \Gamma \vdash \texttt{wf}\ T \qquad \Gamma \vdash \texttt{Mut}(S) <:' \texttt{Mut}(T) \\ \Gamma \vdash S <:' T\end{array}}{\Gamma \vdash S <: T} \quad \text{(SUBTYPING-MAIN)}$$

Figure 2.5: Subtyping rules for reference immutability based on System $\mathtt{F}_{<:>}$.

**Well-formed Types** $\boxed{\Gamma \vdash \texttt{wf}\ T}$

$$\Gamma \vdash \texttt{wf}\ \top \quad \text{(WF-TOP)}$$

$$\Gamma \vdash \texttt{wf}\ \bot \quad \text{(WF-BOT)}$$

$$\Gamma \vdash \texttt{wf Mutable}\ \text{(WF-MUTABLE)}$$

$$\Gamma \vdash \texttt{wf Readonly}\ \text{(WF-READONLY)}$$

$$\frac{X : S..T \in \Gamma}{\Gamma \vdash \texttt{wf}\ X}\ \text{(WF-TVAR)}$$

$$\frac{\Gamma \vdash \texttt{wf}\ S}{\Gamma \vdash \texttt{wf}\ \{x : S\}}\ \text{(WF-RECORD)}$$

$$\frac{\Gamma \vdash \texttt{wf}\ S \qquad \Gamma \vdash \texttt{wf}\ T}{\Gamma \vdash S \to T}\ \text{(WF-ARROW)}$$

$$\frac{\Gamma \vdash \texttt{wf}\ S_1 \qquad \Gamma \vdash \texttt{wf}\ S_2 \qquad \Gamma \vdash \texttt{Mut}(S_1)\ \texttt{<:}\ \texttt{Mut}(S_2) \qquad \Gamma, X : S_1..S_2 \vdash \texttt{wf}\ T}{\Gamma \vdash \forall(X : S_1..S_2).T}\ \text{(WF-ABSTR)}$$

$$\frac{\Gamma \vdash \texttt{wf}\ S \qquad \Gamma \vdash \texttt{wf}\ T}{\Gamma \vdash \texttt{wf}\ S\ \&\ T}\ \text{(WF-INTER)}$$

$$\frac{\Gamma \vdash \texttt{wf}\ S \qquad \Gamma \vdash \texttt{wf}\ T}{\Gamma \vdash \texttt{wf}\ S \mid T}\ \text{(WF-UNION)}$$

$$\frac{\Gamma \vdash \texttt{wf}\ T \qquad \Gamma \vdash \texttt{wf}\ M \qquad \Gamma \vdash \texttt{Mutable}\ \texttt{<:}\ M \qquad \Gamma \vdash M\ \texttt{<:}\ \texttt{Readonly}}{\Gamma \vdash \texttt{wf}\ T\ \texttt{@mut[M]}}\ \text{(WF-MT)}$$

Figure 2.6: Well-formed types in reference immutability.

and `<:'` is the subtyping relation ignoring mutability, which can be considered as the original subtyping rules of Scala.

The well-formed types are also defined in Figure 2.6 to ensure that the mutability in an annotation is always a subtype of `Readonly` and a supertype of `Mutable`, and type parameters do not have bounds with different mutabilities.

### 2.3.3 Typing Rules

The typing for the core language is defined in Figure 2.7.

The typing rule (RECORD-UPDATE) defines the essence of reference immutability: an object can only be modified through mutable references. Attempting to modify the field `b` on the object `a` with type `A @readonly` results in an error stating "trying to mutate a field `b` on readonly `a`".

The field selection is context sensitive, which means the objects referred to by a field from an object with a read-only type should also be read-only. This is known as viewpoint

**Typing and Runtime Typing** $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{(VAR)}$$

$$\frac{\Gamma \vdash \texttt{wf}\ S \qquad \Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda(x : S).t : S \to T} \text{(ABS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{wf}\ S_1 \qquad \Gamma \vdash \texttt{wf}\ S_2 \\ \Gamma \vdash \texttt{Mut}(S_1) =:= \texttt{Mut}(S_2) \\ \Gamma, X : S_1..S_2 \vdash t : T\end{array}}{\Gamma \vdash \Lambda(X : S_1..S_2).t : \forall(X : S_1..S_2).T} \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t : S \to T \qquad \Gamma \vdash s : S}{\Gamma \vdash t(s) : T} \text{(APP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t : \forall(X : S_1..S_2).T \\ \Gamma \vdash S_1 <: S \qquad \Gamma \vdash S <: S_2\end{array}}{\Gamma \vdash t[S'] : T[X \mapsto S']} \text{(T-APP)}$$

$$\frac{\Gamma \vdash t_i : T_i}{\Gamma \vdash \{x_i : t_i \ldots\} : \{x_i : T_i\}\ \&\ \ldots} \text{(RECORD-INTRO)}$$

$$\frac{\Gamma \vdash t : \{x : T\}\ \texttt{@mut[M]}}{\Gamma \vdash t.x : T\ \texttt{@mut[M]}} \text{(RECORD-ELIM)}$$

$$\frac{\Gamma \vdash s : \{x : T\}\ \texttt{@mut[Mutable]} \qquad \Gamma \vdash t : T}{\Gamma \vdash s.x = t : T} \text{(RECORD-UPDATE)}$$

$$\frac{\Gamma \vdash s : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash s : T} \text{(SUB)}$$

Figure 2.7: Typing rules for reference immutability based on System $\mathtt{F}_{<:>}$.

adaptation in existing work [13, 7] and is achieved in the system by letting the mutability of a selection be dependent on the mutability of the receiver. More generally, if an object `a` has type `A @mut[M]`, the type of `a.b` would be `B @mut[M]`, which is shown in the typing rule (RECORD-ELIM). If the field type is already defined with a mutability annotation, the mutability from the receiver is simply attached to the whole type, and the overall mutability is the maximum mutability from all its annotations. Listing 2.4 illustrates the behaviour of viewpoint adaptation.

```
class A:
  var b: B

val a1: A = new A
val a2: A @readonly = a1

a1.b: B
a2.b: B @readonly
```

Listing 2.4: Viewpoint adaptation on different types of references.

In System $\mathtt{F}_{<:>}$, enforcing valid bounds on the declaration site in type abstraction is not required, as noted by [2]. This relaxation allows for fully user-defined subtyping relations. Although users can define conflicting bounds, such as `X : ` $\top..\bot$, instantiating a type checks the lower and upper bounds, ensuring that code with incorrect bounds is not executed and maintaining the type system's soundness. In the system, users are required to define type parameters with the same mutability on the upper and lower bounds, as shown in section 2.2.3. This requirement is enforced by the well-formedness rule (WF-ABSTR) and typing rule (T-ABS).

## 2.4 Other Rules

This section describes other typing rules for reference immutability in Scala that are not covered in the formal model, as well as some additional useful features.

### 2.4.1 Field Initializers and Special Fields

When a field definition is type checked, the `this` reference is considered to be `mutable` in the initializer (the right hand side of the definition) since a field of a class is initialized when the constructor is invoked. Listing 2.5 below shows that giving a mutable type to *this*

is safe as long as it is not escaped or captured. To avoid expensive escaping or capturing analysis, a simple checking is added in the implementation: if the `this` reference is passed to a function or captured by a class or a closure, the compiler issues a warning.

```
class C:
  val self: C = this // safe
  val getC: () => C = _ => this // unsafe capture of the mutable
   reference

val c: C @readonly = ...

c.self: C @readonly
c.getC(): C // danger
```

Listing 2.5: A field initialized to refer to the object itself.

It is useful to keep some fields always mutable even in a read-only reference, making caching and delayed computation possible. Javari also offers a similar feature [18]. When a field is annotated with a concrete `@mutable` annotation, it is referred to as an *always-mutable* field. An always-mutable field can be modified through references with any mutability, and viewpoint adaptation does not apply to the field type. To avoid referring to the read-only part as mutable through an always-mutable field during initialization, the `this` reference is considered to be `readonly` in the initializer of an always-mutable field.

```
class C:
  // danger, can turn a readonly C into mutable
  @mutable var next: C = this

val c1: C @readonly = ...
val c2: C = c1.next
```

Listing 2.6: The danger of mutable `this` inside always-mutable fields.

Listing 2.6 shows the danger of allowing mutable `this` reference in an always-mutable field. If the `this` reference is not considered to be `readonly`, it would be possible to define a field storing the mutable reference. Even if only the read-only reference `c1` is exposed after creation, a user can turn the read-only `C` into a mutable `C`.

A lazy field is a field whose initialization is deferred until the first use. Since the computation of a lazy field can happen at any time after the object is created, it is desired that the read-only part of a read-only reference is not mutated accidentally. Therefore, a similar restriction should be imposed on the initializer of the lazy field, with viewpoint adaptation on the type as well. Specifically, in the initializer of a lazy field, the `this` reference is considered to be `readonly`.

Another way to think about a lazy field is to consider it as a polyread method and some always-mutable fields. Listing 2.7 illustrates a conceptual implementation of lazy fields (the actual implementation of lazy fields in the Scala compiler is more complicated for efficiency and thread safety [17]). Each lazy field can be transformed into two always-mutable fields and a polyread method. Using the lazy field becomes calling the polyread method. The method uses an always-mutable field as a flag to remember if the lazy field has already been initialized. If the flag is `true`, the method returns the value stored in another always-mutable field. Otherwise, the value of the field is computed and stored, the flag is set to `true`, and the value is returned.

```
class C:
  lazy val i: A = <RHS>
  // is desugared into the below:
  @mutable private var value_i: A = _
  @mutable private var flag_i: Boolean = false
  @polyread def i: A @polyread =
    if flag_i then value_i
    else
      val result: A = <RHS>
      value_i = result
      flag_i = true
      result
```

Listing 2.7: A simple implementation of lazy vals.

### 2.4.2 Methods

Although mutability annotations can be attached to types, the syntax of a method definition does not provide any place to express the type of the receiver of the method. Therefore, a mutability annotation is allowed on a method definition itself, and it is interpreted to qualify the type of the method receiver.

```
class C(val next: C):
  var i: Int

  @readonly def m(): C @readonly =
    // unable to mutate i here
    i += 1 // error
    next // the type of next is read-only C
    // the field of read-only this is also read-only

  def n(): C =
```

```
i += 1 // ok
next
```

Listing 2.8: A method showing the behaviour of `this`.

Listing 2.8 shows an example of adding mutability constraint to the receiver of a member function. The method `m` can be invoked on a receiver with a mutability less than or equal to *readonly* (i.e., any mutability level). Inside the method body, the `this` (and `super`) instance is considered to be of type `C @readonly`, so the fields of `this` can be only read but not mutated. In contrast, The method `n` can only be invoked on a mutable receiver.

While the mutability annotations `mutable` and `readonly` can be attached to types and methods, the `polyread` annotation is restricted to member methods, their parameter types, and types used inside the method body. When calling a *polyread* method, the compiler computes the maximum of the mutability of the actual arguments passed for *polyread* parameters, and replaces *polyread* in the method return type with the new mutability. Listing 2.9 illustrates the usage of `polyread` annotations.

```
def f(x: X @polyread, y: Y): Z @polyread = ...

val x: X = ...
val rox: X @readonly = x
val y: Y = ...
val roy: Y @readonly = y
f(x, y): Z
f(x, roy): Z
f(ro, y): Z @readonly
```

Listing 2.9: A *polyread* method and applying it with different mutabilities.

If the `polyread` annotation is applied to a parameter type, it must be placed at the outermost level, as in `List[C] @polyread` rather than `List[C @polyread]`. This placement allows the compiler to locate the `polyread` annotations and compute the mutability at these points simply. If neither the method nor its parameters have `polyread` annotations, the result type and other types inside the scope of the method cannot use the `polyread` annotation. A *polyread* method can be conceptually considered as two methods: one where *polyread* is replaced by *readonly*, and another where *polyread* is replaced by *mutable* as shown in Listing 2.10. The two copies could then be checked separately, and the `polyread` qualifier would never be encountered during any subtype checking. While the overriding version is not supported in the current implementation (the reason is provided in section 2.4.3), the same result can be achieved through a single checking pass.

```
// the f above can be transformed into:
```

```
def f(x: X @readonly, y: Y): Z @readonly = ...
// and
def f(x: X, y: Y): Z = ...
```

Listing 2.10: A possible translation of *polyread* methods.

Javari and ReIm [18, 13] treat `polyread` as a normal mutability between *readonly* and *mutable*. If the usage of `polyread` did not follow the above rules and were treated as a normal mutability between *readonly* and *mutable* in the system, the type system would become unsound. This is because *polyread* annotations from different scopes can represent different mutabilities, as illustrated in Listing 2.11. Previous work does not involve nested methods and classes, thus avoiding the unsoundness issues.

```
// Assume ro_c should be readonly
val ro_c: C @readonly = ...
var i: C @polyread = null
class Op:
  @polyread def set(x: C @polyread): Unit =
    // unsafe, because the polyread of x and i are different
    i = x
  @polyread def get: C @polyread = i
val op: Op = new Op
val ro_op: Op @readonly = op
ro_op.set(ro_c)
val c: C = op.get
// the object refered by ro_c is mutated here
c.x = ...
```

Listing 2.11: Showing `polyread` is not a normal mutability qualifier.

A more robust way to implement the *polyread* annotation is to desugar it into a *mut* annotation with a fresh mutability variable. For example, `def f(x: X @polyread): Y @polyread` can be translated to `def f[M >: Mutable <: Readonly](x: X @mut[M]): Y @mut[M]`. However, this requires modifying the signature of `f` to add the type parameter `M`, and calling `f` without providing the mutability explicitly would require mutability inference, which is currently not supported in the implementation.

While polymorphism of *polyread* is limited at the function level, mutability variables can express more complex mutability bounds and dependencies beyond functions, as shown in the following example:

```
class WrapArray[M >: mutable <: readonly, E, R](a: Array[E] @mut[M]):
  def process[N >: M <: readonly](f: Array[E] @mut[N] => R): R =
    f(a)
```

```
val ss: Array[String] = ...
def isValid: String => Boolean = ...

val readonlyCheck: WrapArray[readonly, String, Boolean](ss)
// can only pass a check function
// that does not mutate the underlying array
readonlyCheck.process[readonly] { as: Array[String] @readonly] =>
  as.forall(isValid)
}

val checkAndMutate: WrapArray[mutable, String, Unit](ss)
// change invalid elements
checkAndMutate.process[mutable] { as: Array[String] =>
  for i <- as.indices do if !isValid(as(i)) then as(i) = ""
}
```

Listing 2.12: An application of mutability variables.

In Listing 2.12, a class `WrapArray` is defined to hide an underlying array. The class has a `process` method that applies a given function to the wrapped array. `M` is a mutability variable that indicates whether the wrapped array can or cannot be mutated by `process`. `readonlyCheck` is an instance of `WrapArray[readonly, String, Boolean]`. The function passed to `readonlyCheck.process` has parameter type `Array[String]@readonly`, so the array can only be read here. In contrast, `checkAndMutate` is an instance of `WrapArray[mutable, String, Unit]`. The function passed to `checkAndMutate.process` is able to read and update an array with type `Array[String]`. The mutability variable `M` has been used across the class and function, which cannot be achieved by the *polyread* annotation.

### 2.4.3 Override and Overload

To override a method in Scala, the overriding method must have the same signature of the parameter types and a result type that is a subtype of the overridden method's result type. This system follows a similar rule. The overriding and overridden members must match with mutability. Specifically,

- the result type or field type of the overriding member must be a subtype of the type of the overridden member;
- the parameter types of the overriding member must be equivalent (`=:=`) to the types of the overridden member, ensuring that they have the same mutability;

- the overriding method must have the same mutability annotation as the overridden method.

In Scala, it is possible to override a field with a method with no parameters. However, this system has different rules for fields and methods, so this kind of overriding is forbidden. To achieve similar behaviour, users can create a method with the same name as the field that simply returns the value of the field. Subclasses can then override this method to simulate overriding a field.

Currently, overloading with types differing only in mutability in Listing 2.13 is not allowed, because the methods have the same signature after being compiled to the target platform (JVM). This feature will be supported in the future by mangling the names automatically.

```
class C:
  // currently, the following overloading is not supported in the
    implementation
  def f(that: C): Unit = ...
  def f(that: C @readonly): Unit = ...
```

Listing 2.13: Overloading with types differing only in mutability.

### 2.4.4 Read-only Classes

In many Scala projects, most classes are intended to be immutable [12]. These classes do not have mutable fields, so the read-only version should exhibit the same behaviour as the mutable version. Rather than requiring the programmer to annotate all uses of a class name as readonly, this can be done once at the class definition site. When defining a class, it can be marked as *readonly* using the `@readonly` annotation, which makes its mutability *readonly* by default, even without any explicit annotation. The compiler checks that a readonly class can only have immutable fields (i.e., `val` rather than `var`), and the mutability of all field types must be *readonly*. All the member methods of a readonly class are automatically annotated with `@readonly` as the receiver mutability.

A readonly class can be extended by a mutable class, but a read-only class can only extend other read-only classes (except `Any` and `AnyRef`, which are handled specially). Therefore, it is safe to override these methods inside a mutable class, as the fields cannot be modified (the overriding methods are also forced to inherit the *readonly* annotation). Listing 2.14 shows an example of defining read-only classes and illustrates the rules.

```
// A is a mutable class
class A

@readonly class B:
  val a: Int // ok
  val b: A @readonly // ok
  val c: A // error, the type of c is not readonly
  var d: Int // error, d is a mutable field

  def f = ... // f is marked as readonly automatically

class C extends B: // C is mutable, but ok
  var e: Int // ok
  override def f = ... // error, f must be annotated with readonly

// error, the parent classes of an immutable class must be immutable
@readonly class D extends C
```

Listing 2.14: An example of read-only classes.

The concept of read-only classes shares similarities with the immutable classes presented in [19], although there are differences. Immutable classes cannot be mutated after creation, and only immutable references can point to immutable objects. In contrast, this design permits mutable references of a subclass to point to an object that is also an instance of a read-only class. If the rule requiring subclasses of a read-only class to be read-only were enforced as well, the definition would be equivalent to that of immutable classes.

## 2.5 Overview of a Class Structure

Listing 2.15 is an overview of a class with different types of fields:

```
@readonly class A:
  val a: A
  val b: Int

class B extends A:
  val c: B
  var d: Int
  @mutable var e: Int
  lazy val f: Int = b
```

Listing 2.15: Coloring different types of fields.

The green-colored fields represent read-only fields that are never changed after creation. The yellow-colored fields are those that may change depending on the mutability of the reference. The always-mutable and lazy fields are colored in red, indicating that their states may change at any time.

Suppose an object is immediately turned into a read-only reference after creation, and its fields are initialized in order (safe object initialization). In this scenario, the fields in green and any objects reachable from them can be safely referred to by other parts of the object. However, the fields in yellow can only be referred to as read-only by the fields in red.

# Chapter 3

# Implementation

In this section, the implementation details of the system inside the *Dotty* compiler are discussed, covering aspects such as the language features, type-checking algorithms, and optimizations made to ensure efficient and accurate checking. The challenges encountered during the implementation process are presented, along with the solutions adopted to address them. By providing a comprehensive overview of the implementation, valuable insights for future improvements and potential adaptations of the system to other programming languages are offered.

## 3.1 Dotty Compiler

The Dotty Compiler, also known as the Scala 3 compiler, is the next-generation compiler for the Scala programming language. It has been designed from the ground up to improve upon the existing Scala 2 compiler. The compiler itself is written in Scala 3, which allows it to fully utilize the new features and capabilities of the language.

In the Dotty compiler, the `Tree` data structure is the core component that represents the abstract syntax tree (AST) of Scala programs. It has several subclasses to represent different types of nodes, such as `Select`, `Apply`, and `Ident`. Each node has a field to store its associated type information. During compilation, the compiler first parses the source code and builds these trees. However, the trees are still untyped. The type-checking phase assigns types to these trees, and they are transformed into typed trees that contain all the necessary type information.

The type information of an expression or a term is represented by the `Type` class. `Type` not only holds the information of possible values but also captures the relations with other types, terms or symbols. Given a variable `val x: List[String]`, the type of `x` is encoded as: `AppliedType(TypeRef(..., "List"), List(TypeRef(TermRef(NoPrefix, "scala"), "String")))`, which means the type reference `String` is applied to a type reference `List`. `"List"`, `"scala"`, and `"String"` in the type are not strings or names. They are symbols referring to the declarations of those entities. When the expression `x` is type-checked, its type is `TermRef(NoPrefix, "x")`, instead of the underlying type `List[String]`. This approach gives developers more information that the value of this expression is the same as the variable `x` holds, without checking the AST. It is possible to obtain the underlying type by widening the `TermRef`.

Listing 3.1 shows the Dotty compiler is organized into three sets of phases: frontend phases, transform phases, and backend phases. The frontend phases deal with parsing, name resolution, and type checking, producing typed trees that are ready for further transformations. The transform phases are responsible for transforming the typed trees into a backend tree. These phases apply various optimizations and transformations to the trees. One of the important phases in this group is the erasure phase, which rewrites the Scala types to the JVM model. The backend phases generate the final bytecode or JavaScript code, depending on the target platform.

```
class Compiler:

  def phases: List[Phase] =
    frontendPhases ::: transformPhases ::: backendPhases

  /** Phases dealing with the frontend up to trees ready for TASTY
   pickling */
  protected def frontendPhases: List[Phase] =
    new Parser ::       // Compiler frontend: scanner, parser
    new TyperPhase ::   // Compiler frontend: namer, typer
    ... ::              // Additional checks and cleanups after type
   checking
    Nil

  /** Phases dealing with the transformation from typed trees to backend
   trees */
  protected def transformPhases: List[Phase] =
    ...                 // Containing transformation phases and erasure
   phase

  /** Generate the output of the compilation */
  protected def backendPhases: List[Phase] =
```

```
    new GenBCode ::     // Generate JVM bytecode
    ... :: Nil
```

Listing 3.1: An overview of Dotty compiler internal.

The overall process of compiling code from Scala source code to JVM bytecode can be shown in Listing 3.2.

```
def runCompile(c: Compiler, source: CompilationUnit): BCode =
  c.phases.foldLeft(source)((cu, p) => p.runOn(cu)).getBCode
```

Listing 3.2: The overall process of compiling.

The `Typer` phase is the most important phase in the compiler, responsible for type-checking an untyped AST and turning it into a typed AST. The `Typer` class contains methods for type-checking various tree nodes, such as `typedSelect`, `typedApply`, and `typedThis` as shown in Listing 3.3.

```
class Typer:
  def typedSelect(tree: untpd.Select, pt: Type)(using Context): tpd.Tree
   = ...
  def typedApply(tree: untpd.Apply, pt: Type)(using Context): tpd.Tree =
   ...
  def typedThis(tree: untpd.This, pt: Type)(using Context): tpd.Tree =
   ...
  ...

  def typed(tree: untpd.Tree, pt: Type)(using Context): tpd.Tree =
    tree match
      case tree: untpd.Select => typedSelect(tree, pt)
      case tree: untpd.Apply => typedApply(tree, pt)
      case tree: untpd.This => typedThis(tree, pt)
      ...
```

Listing 3.3: The structure of the `Typer`.

During the transform phases, a utility phase called `Recheck` can be utilized to recompute and recheck all types in a typed program. This serves as the foundation for phases that refine the original type system with new types and rules, such as the capture check feature [16]. The structure of `Recheck` is similar to Typer. It also contains methods for various tree nodes. By extending the `Recheck` class and overriding certain methods, the developer can implement new rules on these types of trees.

## 3.2 Implementing Mutability

The three mutability qualifiers are defined as classes within the standard library as shown in Listing 3.4. Their subtyping relationships are established through inheritance. The `mut` annotation contains a type parameter with the bound `>: mutable <: readonly`, ensuring that only a mutability can be passed to it. When qualifiers are used directly as annotations (since they are subtypes of `StaticAnnotation`), they are translated into a mutability annotation containing themselves as type arguments. For example, `@readonly` is translated into `@mut[readonly]`.

```
// mutability qualifiers
class readonly extends StaticAnnotation
class polyread extends readonly
class mutable extends polyread
// mutability annotaion
final class mut[M >: mutable <: readonly] extends StaticAnnotation
```

Listing 3.4: Mutability in the standard library.

## 3.3 New Subtyping with Mutability

The `TypeComparer` class is where the subtyping comparison logic is located, and its entry point is `def isSubType(tp1: Type, tp2: Type): Boolean`. Within the `isSubType` function, the mutabilities of the two types are calculated and compared via subtyping first. If the mutabilities follow subtyping, the underlying types are then compared for subtyping, ignoring all mutability annotations. This algorithm follows the subtyping rules outlined in figure 2.5. By isolating most of the mutability-related logic within this single function, the goal is to ensure that the majority of the subtyping comparison logic remains unaffected.

### 3.3.1 Mutability Variable Constraint Handling

As mutability variables are treated as regular type parameters, constraint handling is automatically provided for them. Listing 3.5 demonstrates how the mutability of its arguments is constrained using mutability variables:

```
def f[M1 >: mutable <: readonly,
  M2 >: mutable <: readonly,
  M3 >: M1 <: M2](x: C @mut[M1]): C @mut[M2] =
  // know that M1 <: M2 at this point
```

```
// hence, C @mut[M1] <: C @mut[M2]
x
```

Listing 3.5: Constraint handling on mutability variables.

However, type inference happens during type checking (`Typer`) and, at that phase, the compiler ignores all mutability annotations. Consequently, the `Typer` may infer types with incorrect mutability. Nevertheless, since constraint handling for mutability variables already exists, it is not difficult to modify the `Typer` phase and type inference to account for mutability in the future.

### 3.3.2 Scala Types

In the context of actual Scala programs, it is important to consider additional types that are not explicitly mentioned in figure 2.2:

- `SingletonType`: `SingletonType`s represent types that are guaranteed to contain only a single non-null value. `TermRef`, `ThisType`, `SuperType`, and `ConstantType` are all `SingletonType`s. In most cases, the mutability of these types aligns with the mutability of their underlying types. However, special handling is required for `ThisType` and `SuperType` since the mutability of `this` and `super` depends on the annotations of their enclosing member methods.
- `TypeRef`: `TypeRef` is a reference to a specific type. If it refers to a concrete class, the mutability is determined by whether the class is labelled as read-only by the annotation. If it is a type alias, the mutability is determined by its underlying type.
- `AppliedType`: When checking the subtyping of `AppliedType`'s arguments, a new subtyping check ( `<:` ) needs to be initiated. In this case, the mutability of the type constructor should not affect the mutability of the arguments.

### 3.3.3 Read-only Types by Default

In Scala, primitive types (such as `Int` and `Boolean`) and certain special reference types (like `String`) are inherently immutable. By marking these types as read-only by default, working with them and their members becomes more convenient and seamless.

## 3.4 Recheck for Reference Immutability

After type checking and the majority of transformation phases, a rechecking phase, `CheckMutability`, is introduced to verify the properties of reference immutability. The types of the AST are recomputed and rechecked with the rules for reference immutability. Only during the `CheckMutability` phase are the special subtyping rules for mutability enabled within the `TypeComparer`. The `CheckMutability` phase does not modify the AST, and the generated code only contains mutability annotations, ensuring no runtime overhead is introduced by the feature.

### 3.4.1 Recheck Class Definition

When rechecking a class, trait, or object, the compiler enforces the following rules:

- The type parameters have bounds with the same mutability, as outlined in rule (T-ABS).
- If the class has a `@readonly` annotation:
  - the parent classes must be read-only classes as well (with the exception of `AnyRef`, since it is the default parent for all reference classes and cannot be modified);
  - all fields must be immutable (defined by `val` instead of `var`);
  - the mutability of all field types must be readonly.

### 3.4.2 Recheck Value and Function Definition

To enforce the rules described in Section 2.4.1 and 2.4.2, it is necessary to consider the mutability of `this` or `super` references.

The mutability annotation on a member method determines the mutability of the receiver. Following the type checking phase, each `this` identifier is linked to its corresponding class (like `C.this`). Therefore, the mutability of `C.this` is computed according to the enclosing method that is a member of this class `C`.

The algorithm can be described as follows:

1. Start with the owner symbol of `this` or `super`;

2. If the symbol is a member definition, and the owner of the symbol is the target class, then return the symbol;

3. Otherwise, set the symbol to its owner and repeat the previous step.

Once the member symbol is identified, the type of `this` or `super` becomes the class type with the mutability annotation from the member definition. Except for fields with a concrete `@mutable` annotation and lazy value definitions, the type is the read-only class type. In the case of a symbol representing a nested class, a subsequent search is required, this time with the nested class serving as the target class. The mutability of `this` or `super` depends on the member inside the nested class.

Consider the classes in Listing 3.6: class `B` is nested inside a member of class `A`, and `h` is a non-member nesting function, when checking `A.this` in `h`, the mutability is derived from the annotation of `f`. Similarly, the mutability of `B.this` is derived from the annotation of `g`. It is important to note that the mutability annotations from non-member functions are neither checked nor utilized. Furthermore, if class `C` is a nested class of `A`, the mutability of `C` relies on `A`. Consequently, the mutability of both `this` inside `i` depends on `i`.

```
class A:

  class C:
    def i =
      A.this
      C.this
      // the mutability of both this depends on i
      // since the mutability of C depends on A

  def f =
    class B:
      def g =
        // h is not a member method,
        // never check mutability annotations of h.
        def h =
          A.this // check mutability according to f
          B.this // check mutability according to g
```

Listing 3.6: `this` inside nested classes and functions.

For function definitions, the type parameters must have bounds with the same mutability (T-ABS).

If the body of a function or the return type contains the `@polyread` annotation, the function itself or at least one of its parameters must be annotated with `@polyread`. `polyread`

cannot be treated as a normal qualifier between `mutable` and `readonly`. Adding a mutability variable to each polyread method and rewriting every `@polyread` annotation is difficult to implement and can break compatibility due to change of signature. To avoid the soundness issue, each `@polyread` annotation is bound to its corresponding method symbol internally. As a result, `polyread`s with the same method symbol are subtypes of each other; `polyread`s with different method symbols are not subtypes of each other, as shown in Figure 3.1.

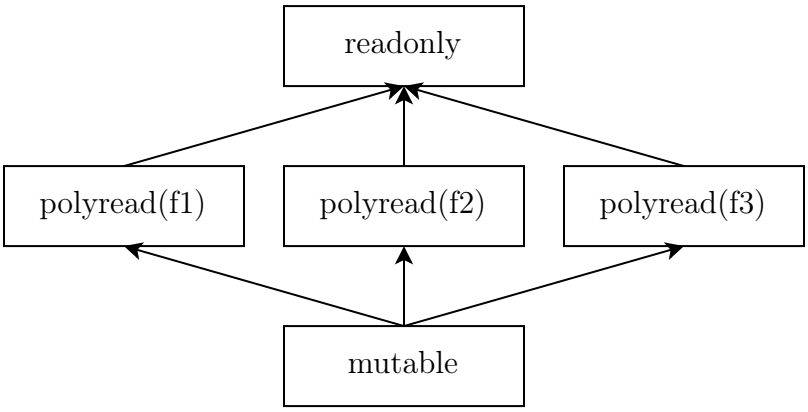


Figure 3.1: The subtype relations of `polyread`s, `mutable`, and `readonly`.

### 3.4.3 Recheck Method Application

For convenience, the receiver of a member method can be considered as an additional parameter in the method signature. As such, the application of member methods follows normal typing rules.

Listing 3.7 demonstrates the rule for *polyread* functions. If the applied method has a `@polyread` annotation or at least one of the parameters has a `@polyread` annotation, the maximum of the mutability is computed from the actual arguments for *polyread* parameters. The `@polyread` in the return type is substituted by the maximum mutability.

```
// a polyread function
def f(a: A @polyread, b: B, c: C @polyread): D @polyread = ...

// M is a mutability variable
val i: A @mut[M] = ...
val j: B = ...
val k: C = ...
```

```
// the first and third parameters have polyread annotation.
// the mutability of the first argument is M,
// the mutability of the third argument is mutable.
// the maximum mutability of these arguments is M | mutable,
// so the result type is D @mut[M | mutable]
f(i, j ,k): D @mut[M | mutable]
```

Listing 3.7: Applying a polyread function.

### 3.4.4 Recheck Pattern Matching

Pattern matching has a similar issue to the one regarding union and intersection types as shown in Listing 3.8. During runtime, it becomes indistinguishable between readonly and mutable references that share the same underlying type.

Despite the fact that a mutable type is considered a subtype of a readonly type, it is inappropriate to transform a readonly type into a mutable one during the matching case. Consequently, when matching a value, the binding variable type of all cases must have mutability greater than or equal to the value type.

```
// a readonly instance of C
val c: C @readonly = ...
c match
  case c: C =>
    c.x = 1 // danger
```

Listing 3.8: Pattern matching on a read-only reference.

## 3.5 Compatibility

The typing rules for reference immutability and the rechecking phase are enabled through a compiler option. It is important to note that no additional code is inserted into the compiled program. If this option is not set, the mutability annotations are ignored and the type checking is performed as in standard Scala. This approach ensures that projects using mutability annotations behave the same way as non-annotated projects when the feature is disabled. In other words, the simple mutability annotations do not change the signatures or affect the compilation process if the feature is disabled.

# Chapter 4

# Evaluation

The experience of adopting reference immutability is presented in this chapter, where the usability, effectiveness and compatibility of this feature is tested. The test suite includes the CS 241E compiler solution, the collection strawman test, and a selection of community projects. These tests demonstrate how the implementation performs in real-world scenarios. To determine the difficulty of using this feature, existing projects are migrated using only the simple mutability annotations: `@mutable`, `@readonly`, and `@polyread`. The results, as shown in table 4.1, are utilized to fix bugs and improve the design.

## 4.1 CS Course Compiler Solution

CS 241E is an advanced second-year compiler course at University of Waterloo, where students are required to develop a bare-bones compiler written in Scala. The majority of the project code is immutable, and almost all classes can be labeled with the `@readonly`

| Item | CS Compiler | Collection | Fansi | discipline |
|---|---|---|---|---|
| Total Lines | 3305 | 653 | 1490 | 441 |
| Changes | 44+, 54- | 215+, 77- | 32+, 32- | 17+, 5- |
| Errors | 25 | 0 | 27 | 4 |
| - Collection Errors | 22 | 0 | 27 | 4 |
| - Other Errors | 3 | 0 | 0 | 0 |

Table 4.1: Table of migrating existing projects.

annotation. Only two classes have mutable state, either through a mutable field or a mutable collection within.

One such example is the `Procedure` class, which represents a procedure tree. It contains a mutable field called `code`, which is initialized after the object is created and mutated during various transformations. To handle this, the class is renamed to `ProcedureImpl` and a type alias is defined: `type Procedure = ProcedureImpl @readonly`. The class type `ProcedureImpl` is only used in places where its mutable field is updated, maintaining read-only references throughout the rest of the project.

The second column labeled "CS Compiler" in table 4.1 shows the result. The size of the CS 241E compiler is around 3300 lines, and the migration modifies 50 lines. There are 25 errors remaining after migration. Because the Scala standard library is not annotated with mutability, most of the error messages reported by the compiler are caused by uses of the standard collections library, for example, calling a member on a read-only reference. Three remaining errors are related to a class in an external library, which could not be annotated.

## 4.2 Collections Strawman

The collections strawman is a self-contained test suite that serves as a testing architecture for possible new designs of the collections framework in the Scala standard library. The strawman code does not reference the existing collections classes in the standard library.

The test `CollectionStrawMan5` in the Dotty compiler test suite is migrated by adding mutability annotations. This code is chosen due to the presence of numerous collection classes with mutable state.

The third column labeled "Collection" in table 4.1 shows the result. The size of `CollectionStrawMan5` is around 650 lines, and the migration modifies 215 lines. There is no compile-time error reported on the ported library

One interesting finding is that adding mutability checking required minor design changes in this library. Specifically, `trait IterableOnce[+A]` is a base trait for all collections. It has a method `iterator` that returns an iterator over the collection. The iterator should be mutable to store the state of the iteration. The method is allowed to be called on any collection (mutable or immutable). Hence, the signature of `iterator` is `@readonly def iterator(): Iterator[A]`, indicating that it can be called on a readonly reference to a collection.

The original design defines `Iterator[+A]` to be a subtype of `IterableOnce[+A]` – this means that an iterator is also itself considered to be a collection. The implementation of the `iterator` method in the `Iterator` class just returns `this`. However, this design does not work when the iterator (the receiver of the method) is readonly: the method then cannot return a mutable reference to it.

To fix the error, `IterableOnce[+A]` is removed from the super classes of `Iterator[+A]`. The method `iterator` in `Iterator[+A]` becomes `def iterator(): Iterator[A] = this`. This new method can only be called on a mutable reference to an iterator.

The higher ratio of changes to total lines in this project compared to others can be attributed to two primary reasons. First, this project requires the design change mentioned above, which introduces modifications across several classes. Secondly, due to the nature of this test, mutations play a significant role in its collections. Consequently, the class definitions cannot be directly annotated as read-only, and each member function requires individual annotations. This comprehensive annotation process contributes to the increased ratio of changes observed in this project.

## 4.3 Other Community Build Projects

The feature is also tested in several other projects within the Scala Community Build, which serves as an extended test suite for the Scala compiler consisting of important open source projects written in Scala. Notably, the `Fansi` and `discipline` projects were ported for this purpose.

The fourth and fifth columns in table 4.1 show the result of those projects. These projects do not extensively rely on mutation, which contributes to the lower ratio of changes to total lines.

To assess the usability of the read-only class feature, a majority of the classes in these projects are annotated as read-only. The remaining errors encountered are primarily related to the usage of the standard collections library within these projects. Despite the annotations applied, the behavior of the annotated projects remains unchanged, meaning they can still be successfully compiled and pass all associated tests, even if the feature is disabled.

## 4.4 Pain Points

Unfortunately, the standard library lacks annotations, leading to a considerable number of errors on commonly used classes like `String`, `Array`, and various collections. To alleviate this issue, a list (in Listing 4.1) of such classes and certain methods is compiled, exempting them from mutability checking during the evaluations.

```
def isReadonlyClass(using Context): Boolean =
  sym.isValueClass
  || sym == defn.CharSequenceClass
  || sym == defn.StringClass
  || sym == defn.Mirror_SingletonClass
  || sym == defn.Mirror_ProductClass
  || sym == defn.Mirror_SumClass
  || sym == defn.EqualsClass
  || sym == defn.ProductClass
  || sym == defn.SerializableClass
  || sym == defn.ThrowableClass
  || sym == defn.ExceptionClass
  || sym == defn.RuntimeExceptionClass
  || sym == defn.ScalaStaticsModuleClass
  || sym == defn.ConversionClass
  || defn.isFunctionSymbol(sym)
  || sym.isClass && (sym.findMutability.isRef(defn.ReadonlyClass))

def relaxApplyCheck(using Context): Boolean =
  val owner = sym.owner
  sym.is(Flags.Synthetic)
  || defn.pureMethods.contains(sym)
  || owner == defn.ScalaStaticsModuleClass
  || owner == defn.OptionClass
  || owner == defn.StringClass
  || sym == defn.Any_asInstanceOf
  || sym == defn.Any_typeCast
  || sym == defn.Array_apply
  || sym == defn.Array_length
  || owner == defn.IterableOpsClass
  || owner == defn.SeqOpsClass
  || owner == defn.IntegralProxyClass
```

Listing 4.1: A list of classes and methods exempted from mutability checking.

An additional challenge encountered is that the mutability checking phase occurs after the type checking phase in the compiler. This ordering presents a problem since type inference takes place during type checking, resulting in potential inaccuracies when the

compiler makes incorrect assumptions regarding type arguments. To overcome this issue, certain sections of the code required manual type annotations to ensure accurate typing results.

# Chapter 5

# Related Work

The concept of reference immutability and its applications have been widely studied, and some mainstream languages have adopted similar concepts. In this section, an overview of the most relevant research and approaches is provided, that have contributed to the understanding and development of reference immutability systems.

## 5.1 Javari and Relm

Reference immutability in Java was first modeled by Birka and Ernst [5] in Javari, an extension to Java that enforces reference immutability. This paper presents a type system, language, implementation, and evaluation of a safe mechanism for enforcing reference immutability, preventing side effects on objects reachable from an immutable reference. Javari specifies immutability constraints using the keyword `readonly` on reference types, methods, and classes. Subsequently, Tschantz and Ernst [18] extended this work with formal type rules based on Lightweight Java and support for Java generics, also introducing syntactic sugar `romaybe` to implement limited parametric mutability methods using overloading.

ReIm [13] focuses on method purity inference, so it has a simpler design compared to Javari. ReIm assigns a single mutability to a whole structure, making it impossible to exclude a field or a type parameter from the immutability check. ReImInfer is the corresponding type inference analysis. The type system is concise and context-sensitive. The type inference analysis is precise and scalable, and requires no manual annotations. In addition, they present an application of the reference immutability type system: method purity inference.

## 5.2 Immutable Generic Java

Immutable Generic Java (IGJ) [19] is a Java extension that incorporates reference immutability into the language without changing Java's syntax. The mutability is encoded as a type parameter of every class. IGJ features three immutability parameters: `Mutable`, `ReadOnly`, and `Immutable`. This system ensures both reference immutability and object immutability simultaneously. Reference immutability guarantees that only mutable references can modify an object, while object immutability ensures that only immutable references point to an immutable object.

## 5.3 roDOT

roDOT [9] is an extension of the Dependent Object Types calculus to support reference immutability. A special marker type with a type member `{` $M : \bot..\top$ `}` is introduced. A type can be combined with the marker type using an intersection type to track its immutability: `T & {` $M : \bot..\top$ `}`. This design makes it possible for a type to depend on the mutability of a reference `x` using the type selection `x.M`. roDOT only presents a calculus that can support an implementation in the Scala compiler, without a complete implementation. If the implementation follows roDOT directly, then the migrated code may become incompatible because the representation of an intersection type with the marker type is different from the original type after compiling.

## 5.4 C/C++ and D

C/C++ utilizes the keyword `const` to indicate that an object or variable is not modifiable. The `const` keyword specifies a variable's value as constant, instructing the compiler to prevent the programmer from altering it. When a member function is declared with the `const` keyword, the object cannot be modified by the function, and it can invoke only `const` member functions. C++ has limited viewpoint adaptation, as reading a field from a `const` object results in a `const` field. However, this rule does not apply to pointers or references.

The D programming language is a general-purpose, statically-typed programming language that prioritizes efficiency, expressiveness, and safety. Designed as a successor to C/C++, D provides comprehensive reference immutability and viewpoint adaptation through

its transitive `immutable` and `const` qualifiers. When an immutable reference is used, any data accessible through it is also immutable, and the same principle applies to the `const` qualifier.

## 5.5 Rust

In Rust, references are either mutable or read-only, and only one mutable reference can exist for any given value, which is achieved by tracking the ownership of an object. This approach enforces safety and helps avoid data races in concurrent programs.

```
let mut s = String::from("hello");

// a reference to a String
// cannot modify underlying data through it
let r1 = &s;
r1.push_str(", world"); // error

// a mutable reference to a String
// borrow `s` as mutable
let r2 = &mut s;
r2.push_str(", world"); // ok

// a mutable reference to a String
// error, cannot borrow `s` as mutable more than once at a time
let r3 = &mut s;

println!("{}, {}", r2, r3);
```

In the provided code snippet, a mutable `String` named `s` is instantiated. Following that, a read-only reference named `r1` is defined, pointing to the same `String` object. Attempting to modify the underlying data through `r1` is forbidden and leads to a compile-time error.

Subsequently, a mutable reference named `r2` is created by borrowing the mutability of `s`. This declaration enables modifying the underlying data through `r2`, while still preserving the read-only nature of `r1`.

However, attempting to create an additional mutable reference `r3` to the same `String` and then printing both `r2` and `r3` simultaneously using `println` on the last line would result in a compile-time error in Rust. This error serves as a preventive measure, disallowing the existence of multiple mutable references to the same value simultaneously.

In Rust, references can be qualified by lifetime parameters, but the mutability aspect remains concrete and fixed. As a result, it is not possible to express mutability polymorphism in the language.

# Chapter 6

# Conclusion

In this thesis, I have addressed the issue of reference immutability in Scala, a multi-paradigm programming language that combines functional and object-oriented programming. While Scala promotes immutability through features like immutable variables and collections, it lacks a built-in mechanism for controlling and enforcing reference immutability.

I have presented a system of reference immutability in Scala with a formal model for the core language. The system is designed to handle the complexities of the Scala type system. Mutabilities are encoded as types inside annotations, and mutability operations are encoded using union and intersection type operations. This design allows for the expression of flexible mutability constraints and enables checking mutabilities through subtyping comparisons. I have carefully designed the new types to ensure backward compatibility both before and after a project's migration to this feature.

One of the key challenges addressed is handling nested classes and functions, which is ignored in previous work. I developed a more general design for viewpoint adaptation and an algorithm to derive appropriate mutability of a reference based on its enclosing scope.

I have implemented the feature in the Dotty compiler, closely following the rules described in the formal model. The changes to the compiler are encapsulated within a single module to minimize their impact on other parts of the compiler.

The design presented in this thesis establishes a foundation for incorporating arbitrary constraints on any types in Scala. When developing a new feature that requires constraints, developers can simply define the constraints as classes and implement the corresponding rules in a new phase by checking them using subtyping. One advantage of this design

is that the annotation mechanism and handling of nested code can be reused, without worrying about backward compatibility.

In conclusion, my work on reference immutability contributes an expressive and useful design to control and enforce reference immutability by extending the type system. This feature promotes more predictable behavior and prevents unintended mutations, reducing the risk of bugs and making code easier to reason about.

# References


[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1985. ISBN 0262010771.

[2] Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 666–679, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346603. doi: 10.1145/3009837.3009866. URL https://doi.org/10.1145/3009837.3009866.

[3] Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. *SIGPLAN Not.*, 52(1):666–679, jan 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009866. URL https://doi.org/10.1145/3093333.3009866.

[4] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 249–272, 2016.

[5] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, page 35–49, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138318. doi: 10.1145/1028976.1028980. URL https://doi.org/10.1145/1028976.1028980.

[6] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. Exploring language support for immutability. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 736–747, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884798. URL https://doi.org/10.1145/2884781.2884798.

[7] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 28–53, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73589-2.

[8] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. *SIGPLAN Not.*, 48(1):287–300, jan 2013. ISSN 0362-1340. doi: 10.1145/2480359.2429104. URL https://doi.org/10.1145/2480359.2429104.

[9] Vlastimil Dort and Ondřej Lhoták. Reference Mutability for DOT. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:28, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-154-2. doi: 10.4230/LIPIcs.ECOOP.2020.18. URL https://drops.dagstuhl.de/opus/volltexte/2020/13175.

[10] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 21–40, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315616. doi: 10.1145/2384616.2384619. URL https://doi.org/10.1145/2384616.2384619.

[11] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. *SIGPLAN Not.*, 47 (10):21–40, oct 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384619. URL https://doi.org/10.1145/2398857.2384619.

[12] Philipp Haller and Ludvig Axelsson. Quantifying and explaining immutability in scala. *Electronic Proceedings in Theoretical Computer Science*, 246:21–27, apr 2017. doi: 10.4204/eptcs.246.5. URL https://doi.org/10.4204%2Feptcs.246.5.

[13] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim and reiminfer: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 879–896, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315616. doi: 10.1145/2384616.2384680. URL https://doi.org/10.1145/2384616.2384680.

[14] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3): 396–450, may 2001. ISSN 0164-0925. doi: 10.1145/503502.503505. URL https://doi.org/10.1145/503502.503505.

[15] Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. Encoding featherweight java with assignment and immutability using the coq proof assistant. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, page 11–19, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312721. doi: 10.1145/2318202.2318206. URL https://doi.org/10.1145/2318202.2318206.

[16] Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondřej Lhoták. Safer exceptions for scala. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*, SCALA 2021, page 1–11, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391139. doi: 10.1145/3486610.3486893. URL https://doi.org/10.1145/3486610.3486893.

[17] Scala. Lazy vals initialization, 2022. URL https://docs.scala-lang.org/scala3/reference/changed-features/lazy-vals-init.html. Accessed on May 1, 2023.

[18] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to java. *SIGPLAN Not.*, 40(10):211–230, oct 2005. ISSN 0362-1340. doi: 10.1145/1103845.1094828. URL https://doi.org/10.1145/1103845.1094828.

[19] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kie|un, and Michael D. Ernst. Object and reference immutability using java generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 75–84, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938114. doi: 10.1145/1287624.1287637. URL https://doi.org/10.1145/1287624.1287637.