École Polytechnique Fédérale de Lausanne

Macro Annotations for Scala 3

by Zhendong Ang

Master Thesis

Approved by the Examining Committee:

Prof. Martin Odersky
Thesis Advisor

Dr. Fengyun Liu
External Expert

Nicolas Stucki
Thesis Supervisor

EPFL IC IINFCOM LAMP
INR 319 (Bâtiment INR)
Station 14
CH-1015 Lausanne

June 17, 2022

Abstract

Macro annotations are an important feature in Scala 2 macro system. Many projects use macro annotations to implement their systems or libraries. Due to the unportability of Scala 2 macro system, Scala 3 redesigns the macro system to make it more reliable and portable. But macro annotations have not been implemented in Scala 3 yet, resulting in some inconvenience in migrating projects that use macro annotations to Scala 3.

Following the core design of Scala 2 macro annotations, we introduce the macro annotations as transformations from definitions to definitions. In this thesis, we also list rules to keep our simplicity but not hurt its functionality at the same time. The main difference between macro annotations of Scala 2 and Scala 3 is that in Scala 2 macro annotations expand before typechecking, while they expand after typechecking in Scala 3.

Our implementation is based on a breadth-first approach, which helps us implement a tail-recursive transformation. We write test cases covering common and even uncommon cases to test the correctness of our macro annotation system.

Contents

Chapter 1

Introduction

1.1 Motivation

Macro annotations are an extension of Scala 2 macro system [1]. It was implemented in the macro paradise plugin from Scala 2.10.x to 2.12.x and available in Scala 2.13 with the `-Ymacro-annotations` flag. However, Scala 3 redesigned the metaprogramming system and macro annotations are not supported yet. But now, there are demands for macro annotations in Scala 3.

Macro annotations expand definitions into definitions. For example, here is a classical example `@memoize`:

```scala
@memoize
def fib(n: Int): Int =
  if (n <= 1) 1
  else fib(n - 1) + fib(n - 2)
```

Listing 1.1: Original code

```scala
val fibCache = Map.empty[Int, Int]
@memoize
def fib(n: Int): Int =
  if fibCache.contains(n) then
    fibCache(n)
  else
    val res =
      if (n <= 1) 1
      else fib(n - 1) + fib(n - 2)
    fibCache(n) = res
    res
```

Listing 1.2: After expansion

The macro annotation `@memoize` generates a cache and changes the body of the function to first check if the result has been cached. It makes programming more efficient.

Some other demands emerge when users try to migrate their project to Scala 3, which uses macro annotations in Scala 2. Sometimes, it is challenging to finish the same job using normal macros. ScalaPy, which ports Python libraries to Scala, is an example [2]. It uses macro annotations to create facades for Python objects. Without macro annotations, migrating to Scala 3 faces many problems, and metaprogrammers have to use tortuous ways to fix them.

1.2 Objective and Results

The object of this project is to implement macro annotations in Scala 3. Following the design in Scala 2, the goal is to support a transformation from definitions to definitions. The implementation successfully provides a way to process the transformation and insert new generated definitions in proper places. Tests show that the new macro annotation system works in many cases. The only exception is adding trees to the top-level.

1.3 Thesis Structure

In Chapter 2, we introduce some technical background that helps to understand this thesis, including compiler phases, abstract syntax tree representation, and some tools in Scala 3 metaprogramming.

In Chapter 3, we present the design idea of the macro annotation system. It lists some rules that the users can do and cannot do with macro annotations.

In Chapter 4, we explain how we implement the transformation. It covers an efficient, tail-recursive approach to transform the definitions and other details.

In Chapter 5, we list all the test cases that have been covered. In Chapter 6, we review the related works and discuss the future work.

Chapter 2

Background

2.1 Compiler Phases

The Scala 3 compiler consists of a list of phases. Phases can manipulate the Abstract Syntax Tree (AST) and generate information [3]. Currently, all the phases are split into four subgroups: frontend, pickler, transform, and backend. Unlike the traditional way that each phase needs to traverse the whole AST, one subgroup only traverses it once. There are some phases that we care about: the `typer` phase is in the frontend group and the inlining phase is in the pickler group.

2.2 Definitions in the AST

In this section, we present some definitions and their representation in the AST.

- Values and methods are represented as `ValDef` and `DefDef` nodes.
- Classes, traits and types are all represented as `TypeDef` nodes. But
  - For classes and traits, the right-hand side is a `Template` node, which contains information of the class: constructor, parents, and body.
  - For types, the right-hand side is a type alias or a `TypeBounds` node.
- Objects are represented as pairs of a `ValDef` and a `TypeDef` for class.

```
// original code
object Foo
```

```
final lazy module val Foo: Foo = new Foo()
final module class Foo() extends Object()
```

2.3 Scala 3 Metaprogramming

Scala 3 redesigns the metaprogramming system [4]. We present some of the new features that are related to our thesis. Quotes and Reflections are two important tools to create definition trees.

2.3.1 Quotes and Splices

Scala 3 macros are built on these two fundamental operations: quotes and splices [5, 6]. Quotes convert code to tree representations, and splices go the opposite way, converting tree representations to program code. So, quoting definition code is an efficient way to obtain a definition tree. Another difference between quotes and splices is that quotes delay the execution, but splices evaluate immediately and splice the result into surrounding expression. Thus a macro annotation in quotes should delay expansion, which will be discussed in section 4.2.2.

Level is an important concept, which is defined as a count of the number of quotes minus the number of splices surrounding an expression or definition. For example:

```
// level 0
'{ // level 1
  var x = 0
  ${ // level 0
    x += 1
    'x // level 1
  }
}
```

The level is maintained by `StagingContext` in the compiler. We use it to determine whether a macro annotation is directly inside a quote.

2.3.2 Reflection

Reflection is a wonderful tool to analyze and construct typed AST. It is defined in `quotes.reflect`. It provides two ways to construct definitions: apply and copy. Here is an example of `ValDef`:

```
def apply(symbol: Symbol, rhs: Option[Term]): ValDef
def copy(original: Tree)(name: String, tpt: TypeTree, rhs: Option[Term]): ValDef
```

Apply method creates a new definition with a given symbol, which specifies the signature, and the right-hand side. The copy method uses an existing definition but changes its properties.

2.3.3 Example

Here we give an example of using quotes and reflections to create definition trees in macro annotations. The following code is one of the implementations of the macro annotation `@memoize` discussed in section 1.1. This implementation is not the best one but can be used to show how these two tools, quotes and reflections, are employed. We will discuss the API of macro annotations in section 3.3.

We use quotes to create the right-hand side of the definitions and reflections to create the definition trees.

```scala
class memoize extends MacroAnnotation {
  override def transform(using Quotes)(tree: Definition): List[Definition] =
    import quotes.reflect._
    tree match
      case DefDef(name, params, tpt, Some(fibTree)) =>
        // right-hand side of value fibCache
        val cacheRhs = '{Map.empty[Int, Int]}.asTerm
        val cacheSymbol = Symbol.newVal(tree.symbol.owner, "fibCache",
          TypeRepr.of[Map[Int, Int]], Flags.EmptyFlags, Symbol.noSymbol)
        // val fibCache tree. Use apply API
        val cacheVal = ValDef(cacheSymbol, Some(cacheRhs))
        val fibCache = Ref(cacheSymbol).asExprOf[Map[Int, Int]]
        val n = Ref(params.head.params.head.symbol).asExprOf[Int]
        // right-hand side of def fib
        val rhs = '{
          if $fibCache.contains($n) then
              $fibCache($n)
          else
            val res = ${fibTree.asExprOf[Int]}
            $fibCache($n) = res
            res
        }.asTerm
        // def fib tree. Use copy API
        val newFib = DefDef.copy(tree)(name, params, tpt, Some(rhs))
        List(cacheVal, newFib)
}
```

Chapter 3

Design

In the chapter, we present the design details of macro annotations. It includes the phase of macro annotation expansion in the compiler, the rules of macro annotation expansion, and the API of macro annotation expansion for users.

3.1 Macro Annotations Phase

The key distinction between macro annotations in Scala 2 and 3 is when the macro annotations expand. In Scala 2, macro annotations expand before the typechecking. Thus, they take untyped definitions and transform them into untyped definitions. However, users' creation of untyped, or syntactic, ASTs, not typed, or semantic, ASTs leads to some issues. During the typechecking, the compiler could accidentally make mistakes on implicit resolution, overload resolution, etc. So, in Scala 3, the whole metaprogramming system is built semantically driven. Macro annotations should expand after the typechecking, and all the generated trees are well-typed. As a result, the users will notice that some previously correct use of macro annotations in Scala 2 is not allowed in Scala 3 anymore. We take the test case from `sbt-example-paradise` as an example, as shown below, where the macro annotation generates a definition `hello` inside the annotated object. The piece of code compiles in Scala 2 but not in Scala 3. The typer will throw an error showing that `hello` is not a member of object `Test`.

```scala
@hello object Test {
  // @hello generates:
  // def hello: String = "hello world"
  println(this.hello) // OK
}
```

Listing 3.1: In Scala 2

```scala
@hello object Test {
  // @hello generates:
  // def hello: String = "hello world"
  println(this.hello) // error
}
```

Listing 3.2: In Scala 3

For the precise position of the macro annotation phase, we decide to mix it with the inlining phase [7], instead of regarding it as a separate phase. Because the users may call an inline method in a generated definition, such inlining would be ignored if we put the macro annotation phase after the inlining phase. Moreover, the transformation structures of these two phases are similar, so merging them is easy to implement. In the remaining part of this thesis, we will use the term inlining phase to refer to the mixed phase of inlining and macro annotation transformation.

3.2 Macro Annotation Transformation

The basic idea of macro annotations is to expand an annotated definition into definitions. The output definitions will replace the input definition after the expansion. In other words, the transformation rule is to say that a macro annotation can change the original definition into another typed definition, and/or generate new definitions around the annotattee. The typed annottees can be arbitrary definitions, such as classes, objects, traits, functions, types, values, and variables.

3.2.1 Discussion about one other more powerful transformation

The transformation design follows the principle in Scala 2. Previously, we designed and implemented a more powerful transformation, which allows macro annotations to insert new definitions into any owner of the annotattee. For example, as shown below, annotation `foo` can not only insert definition trees into class `B` but also class `A`.

```
class A:
  class B:
    @foo
    def f(x: Int) = x + 1
```

There are two reasons why we discarded this design. First, the insertion of new trees into other places, not just around the input definition, leads to a more complicated implementation and increases overheads, since the transformer needs to first collect and store all the new trees, then insert them, instead of simply replacing the original annottee. Furthermore, it is hard to think of a practical test case in this way. Even if there is such a need, the users can avoid programming in this way. They should annotate class `B` instead of method `f` in the above example. As a result, we decide to follow the Scala 2 design to keep it simple and clean.

3.2.2 Special Cases

Besides all the normal transformations, there are two special cases for method parameters and objects.

Method Parameters Method parameters discussed here include type parameters and value parameters. If the annottee is a method parameter, the input definition is the parameter itself and its owner, and the output definitions will replace the owner method. For example, if the code is `def foo(@f x: Int) = x + 1`, the input is `x` and `foo`, and the output will replace the owner method `foo`, instead of the parameter `x`.

Object As mentioned in section 2.2, the object is represented as a value and class definition. Annotating an object leads to the annotation of both the value and the class definition. The macro annotation will not expand separately for the two definitions but will take both definitions as the input and generate definitions to replace them.

3.3 API design

The macro annotation is defined as a trait.

```
trait MacroAnnotation extends StaticAnnotation {
  def transform(using Quotes)(tree: Definition): List[Definition] =
    report.errorAndAbort(tree.show, tree.pos)
  def transformObject(using Quotes)(valTree: ValDef, classTree: TypeDef):
    List[Definition] =
    report.errorAndAbort(classTree.show, classTree.pos)
  def transformParam(using Quotes)(paramTree: Definition, ownerTree: Definition):
    List[Definition] =
    report.errorAndAbort(paramTree.show, paramTree.pos)
}
```

It has three different transform methods, corresponding to annotations of normal definitions, method parameters, and objects, as discussed in section 3.2. They take one or more definitions as input and output a list of definitions. The users will write their macro annotation by extending `MacroAnnotation` and implementing the transform methods.

3.4 Other Rules

In this section, we will give some other rules illustrating what the users can do and cannot do with macro annotations.

3.4.1 Orders of Multiple Macro Annotations

One definition annotated by multiple macro annotations As for a definition annotated by several macro annotations, such as `@f @g def foo(x: Int) = x + 1`, the first macro annotation will expand last, while the last will expand first. Exactly, there is no preference for from which direction the macro annotations expand. It is just more natural to regard the definition as `@f {@g def foo(x: Int) = x + 1}`.

Notice that it is not encouraged to write one macro annotation relying on another which annotates the same definition. If so, the user could write a new macro annotation that does the job of these two dependent macro annotations.

Nested definitions both annotated by macro annotations In this case, we follow the design in Scala 2 in that the outer macro annotation expands first, then the inner one. Such an order brings an advantage that the inner macro annotation can access the definitions generated by the outer macro annotation. For example, as shown below, macro annotation `@g` can call `bar` generated by macro annotation `@f`.

```
@f class A:
  @g def foo(x: Int) = x + 1
```

Listing 3.3: Original Code

```
def bar(x: Int) = ??? // generated by @f
@f class A:
  // changed by @g
  @g def foo(x: Int) = bar(x) + 1
```

Listing 3.4: After Macro Expansion

3.4.2 Refer to the generated definitions

In one macro annotation, it can, of course, access the definitions generated by itself. We should consider that the generated definitions may also be annotated by some macro annotations, which may generate some other definitions. To be clear, here is an example. Method `v` is generated by macro annotation `@f` and can be accessed by method `u`. Method `w` is generated by macro annotation `@g` and can be accessed by method `v`.

```
@f def foo(x: Int) = x + 1
```

Listing 3.5: Original Code

```
def w(x: Int) = ??? // generated by @g
@g def v(x: Int) = w(x) // generated by @f
@f def u(x: Int) = v(x) + 1
```

Listing 3.6: After Macro Expansion

But is it possible that method `u` accesses method `w`? We decide no. The first reason is that it is not commonly used and can be avoided by writing a new macro annotation annotating method `u` that generates both methods `v` and `w`. The second reason is that if we limit the accessibility of macro annotations only to directly generated definitions, such as `v`, not `w`, it could make the transformation more efficient, which will be discussed in section 4.2.

3.4.3 Change the annotation of the same annottee

Here is an example for this case. We have a method `foo` which is annotated by macro annotations `@f` and `@g`: `@f @g def foo = ???`. `@f` tries to add a new macro annotation `@h` to this method and delete `@g`, expanding the method to `@f @h def foo = ???`.

Our design does not support such changes. A macro annotation adding or deleting other annotations of the same annottee leads to undefined behavior. The primary reason is that it violates the principle that one macro annotation should not be dependent on another if they annotate the same definition. Users can always write a new macro annotation to finish the job if they need to add or delete a macro annotation into or from the annottee. The other reason is that it would make the implementation more complicated. We need to keep a working list of macro annotations during the transformation. In some extreme cases, if a later macro annotation deletes a processed macro annotation, we need to revert the transform. Actually, in our implementation (section 4.2), we first collect all the macro annotations of a definition, then expand them one by one.

Since disallowing such change does not hurt the functionality, we prefer to keep the design simple.

Chapter 4

Implementation

In this chapter, we provide details of the implementation of our macro annotation transformation. We first discuss how the `TreeMap`, which transforms the trees recursively, is established. Then we explain how to transform a single annotated definition.

4.1 AST Transformation

In the mixed inlining and macro annotation phase, we transform trees using `TreeMap` defined in `trees.scala`. It traverses trees recursively by their structure. We implement our `TreeMap` by extending it. Because we merge the macro annotation phase with the inlining phase, we just make changes on `InliningTreeMap`.

4.1.1 Method transform and Method transformStats

In the `TreeMap`, we only care about two methods: `transform` and `transformStats`. We override them to handle macro annotation expansion.

method `transform` Method `transform` transforms a single tree. If the tree is a definition, it will be processed for macro annotation expansion in the next step. To make the code cleaner, we implement macro annotation expansion of a definition tree in a separate object `MacroAnnotationTransformer`, which will be discussed in section 4.2.

Among the definitions, we treat parameters differently. Considering that `transformParam` API takes the parameter and its owner tree as input, we will handle the annotated parameter when we process its owner tree because obtaining the tree from the owner's symbol is not reliable. Thus, when the transform method handles a parameter, which can be determined by flags, it will skip macro annotation expansion.

method `transformStats` Method `transformStats` transforms a list of statements, which is often the body of a class or a part of a block. Definitions locate in a list of statements in most cases. In the original inlining phase, this method is not overridden. But macro annotation phase requires a small change to handle objects. The overridden `transformStats` still keeps other logic, such as conservative map, and context handling.

Recall that an object is represented as a value and a class definition in AST. The two definitions are always part of a list of statements. In macro annotation transformation, we need to handle the two definitions together. So in the overridden `transformStats`, pairs of object value and class definitions are identified and combined in a `Thicket`. Then object pairs can be treated as single definitions.

4.1.2 Merging into Inlining Phase

Success in mixing the two phases relies on that the inlining phase pays little attention to the definitions, but the macro annotation phase almost only cares about the definitions. The inlining phase inlines the call to inline methods, which only exist in the deeper structure, such as the body, of a definition. The macro annotation phase handles the annotation of a definition first, then jumps into its deeper structure.

As a result, when transforming a definition, we first transform it in `MacroAnnotationTransformer` and then call the `transform` method recursively on the deeper structure of the output definitions.

4.2 Tail-recursive Single Definition Transform

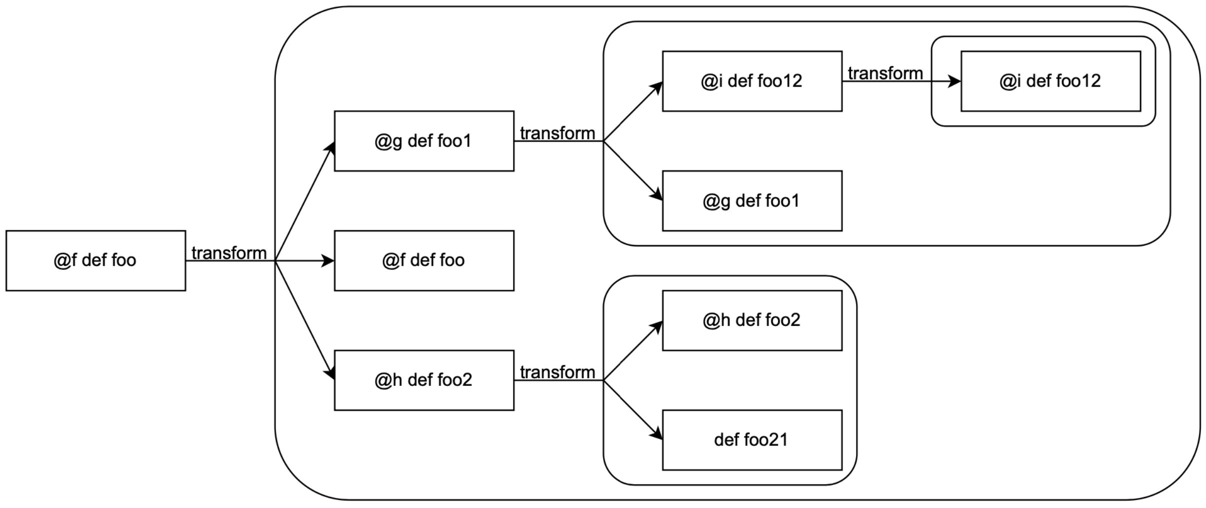
In this section, we explain the core part of the macro annotation transformation, which is defined in the object `MacroAnnotationTransformer`. It takes a single definition as input and returns a list of definitions.

4.2.1 Structure of the Single Definition Transformation

Our job is not only to expand all the annotations of the input definition, because new generated definitions may also be annotated by macro annotations.

Depth-First Approach A natural way is to transform the new generated definitions one by one recursively, then combine the results. However, thanks to our design which limit access only to direct generated definitions, we could think of a breadth-first approach that is tail-recursive and efficient.

Figure 4.1: Depth-First Approach



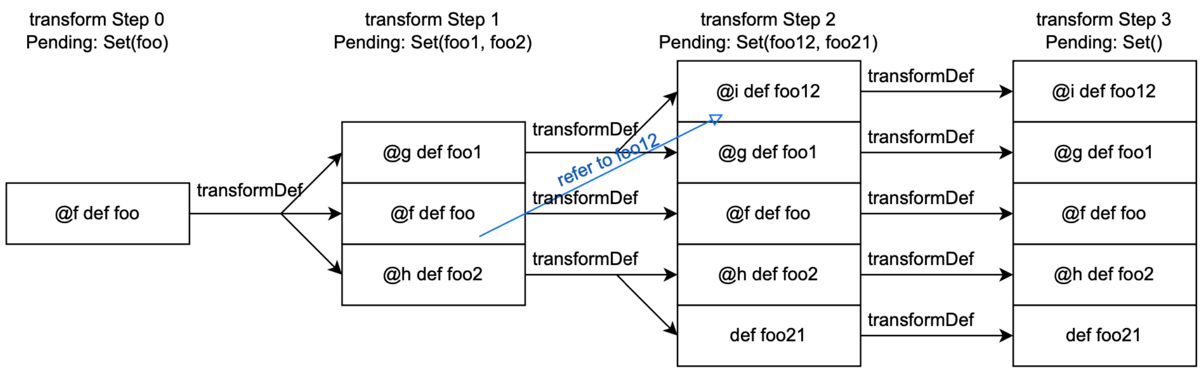
Breadth-First Approach As shown in Figure 4.2, the transform takes a list of definitions as input, instead of a single definition. The input is a list containing only one element. In each step, we expand the annotations of every definition that has not been processed by a method `transformDef` and construct a new list with the results as the output, which is also the input of the next step, until all definitions are processed. Method `transformDef` works similarly to the `transform` method in the Depth-First Approach, which takes one definition and expands it to a list of definitions, but it does not expand the annotations of the new generated definitions. So this method is not recursive.

```
object MacroAnnotationTransformer:
  @tailrec
  def transform(defns: List[Tree], pending: Set): List[Tree] =
    if pending.size == 0 then defns
    else
      transform(defns.flatMap(transformDef), updatedPending)

  // non-recursive
  def transformDef(defn: Tree, pending: Set): List[Tree] = ...
```

The only overhead is that we need to maintain a set to record the symbol of the definitions that have been processed. The transformation can be tail-recursive.

Figure 4.2: Breadth-First Approach



There is no worry about the access to trees generated in the latter steps. For example, method `foo` call `foo12` in step 1, shown as the blue arrow in Figure 4.2. Our design rules have decided that such access is invalid because method `foo12` is not directly generated by the annotation `@f` of method `foo`.

4.2.2 Method `transformDef`

Method `transformDef` expands all the annotations of the input definition and returns a list of definitions. It simply ignores the annotations of the generated definitions.

The logic of this method is simple. We first collect all the annotations of the input definition and its parameters if it is a method. It is correct because we do not allow users to change annotations during the expansion. Then for each annotation, it instantiates the annotation, applies the `transform` method with corresponding parameters, and handles the result by collecting the new generated definitions and entering new symbols.

```
def transformDef(defn: Tree, pending: Set): List[Tree] =
  if !(pending.contains(defn.symbol)) then return List(defn)
  if level > 0 then return List(defn) // level check
  val annots = // collect all the annotations of the input tree
             // and the parameters if the input is a method
  var newTreesBefore, newTreesAfter // collect new generated definitions
  if annots.isEmpty then return List(defn)
  val newDefn = annots.foldLeft(defn){(defn, ann) =>
    val instance = // instantiate annotation
    val result = instance.transform(defn, ...)
    val newDefn = // find the changed annottee in the result
    // add new generated trees into newTreesBefore and newTreesAfter
    // enter new symbols into its owner's scope
    newDefn
  }
  newTreesBefore ++ List(Defn) ++ newTreesAfter
```

Then we give details of three issues in this method:

Level Check Users may commonly use quotes to construct new definitions. Those definitions might be annotated with macro annotations. We do not expect these annotations to expand when compiling the quotes. Expansion should be delayed and performed after the definitions have been inserted where they should be.

We use staging context and level check to avoid incorrect expansion. Recall that level in the context being larger than 0 means the tree is directly inside a quote and will be inserted somewhere else. So we can just ignore macro annotation expansion here and return the original annottee.

Instantaite Macro Annotations We get inspiration from the interpreter class of `Splicer.scala`. It implements methods to interpret many kinds of trees. When we transform an annottee tree, its annotation also has a tree in the form:

```
// instantiate the annotation by calling the constructor with args
Apply(Select(New(annot), <init>), args)
```

One of the methods `InterpretNew` is just what we need to instantiate macro annotations. We can simply construct our interpreter by extending the existing one and reusing the code.

Enter new symbols When a new definition is created, its symbol should be added into its owner's scope if its owner is a class. There are two cases for entering new symbols:

- New generated definition which will be placed around the annottee.
- If the annottee is a class, its body may get changed. We need to check the definitions in its body one by one to determine whether a new definition is added.

Chapter 5

Testing

In this chapter, we will give descriptions of the test cases that have been covered.

5.1 Positive Tests

- Change the body of functions, types, values, and variables
- Add definitions into the body of classes, objects, and traits
- Macro annotations on types and value parameters
- Insert definitions before the annottee and access them inside the body
- Generate definitions annotated by other macro annotations
- Macro annotations in splices
- Macro annotations in quotes
- Multiple macro annotations on the same annottee
- Nested definitions both annotated by macro annotations

5.2 Negative Tests

- Macro annotations that access indirectly generated definitions
- Macro annotations that access definitions generated by latter expanded annotation
- Macro annotations that change the signature of a method using reflection copy API

- Macro annotations that change the parents of a class using reflection copy API
- Macro annotations that change the type of a value using reflection copy API

Chapter 6

Related and Future Work

6.1 Related Work

Macro Annotations in Scala 2 Even though Macro annotations in Scala 2 are based on the metaprogramming system in Scala 2, which is different from the system in Scala 3, our work gets many design inspiration from it. The most significant difference between macro annotations in Scala 2 and 3 is that, as mentioned before, macro annotations are expanded before typechecking in Scala 2, which means that it transforms untyped definitions into untyped definitions. The whole system is not semantically driven and may cause accidental mistakes in typechecking. The untyped nature of macro annotations in Scala 2 also reduces the tools available to the users, who may lack much type information [1].

Nemerle Nemerle is a programming language based on .NET platform. It extends the attribute feature of .NET by making it possible to transform definitions [8]. Nemerle also inspires the design of macro annotations in Scala 2 [1].

Derive Macros in Rust Derive macros work similarly to macro annotations. It takes the token stream of a structure and returns a set of new items, which will then be appended to the block that the input token stream is in [9]. Compared to our macro annotations, derive macros have restricted functionality. It cannot generate and transform definitions in arbitrary patterns.

6.2 Future Work

Add Definitions to Top-Level Scala 3 has dropped the package objects feature [4, 10]. Now users can write all definitions at the top-level. To make it possible, the compiler creates a synthetic object `filename$package` to hold some certain definitions, including value, type, method, etc.

During macro annotation expansion, if users desire to add definitions to top-level and the synthetic object does not exist, the compiler needs to create one. But technically, we lack the infrastructure to create an object in this phase now.

Reflection `ClassDef` API is limited Metaprogrammers can use `ClassDef` API, which is now experimental, to create new classes. Currently, however, its functionality is limited in that it only supports creating classes without parameters. Future work for further improvement is required.

Issues about Reflection `copy` APIs Many users use reflection `copy` APIs to change existing trees. Reflection `copy` APIs provide a way to change more than the body, or the right-hand side of a definition. For example, the `copy` API for `ValDef` is shown below. It allows users to change the name or the type of a value definition.

```
def copy(original: Tree)(name: String, tpt: TypeTree, rhs: Option[Term]): ValDef
```

We found two issues related to these APIs.

In the first example, the macro annotation changes the type and the value of the value `a`. An error is thrown during compilation. After inspecting ASTs after each phase, we find that the type is indeed when the macro annotation expands, but the original type is reverted after several phases, which causes the inconsistence between the type and the right-hand side value. The problem may be caused by `copy` APIs not creating new symbols, which store type information. So the compiler obtains original type information from the unchanged symbol. The same problems also happen in other cases, such as signatures of methods, parents of classes, and bounds of type parameters. This issue requires us to reflect on the design or the implementation of the `copy` APIs.

```
// Original code
@toInt def a: String = "3"
// After inlining
@toInt def a: Int = 3
// After several phases
@toInt def a: String = 3.asInstanceOf[String] // error
```

Another test case seems to imply that a type checking case is missed after the inlining phase. The macro annotation `upper` changes the upper bound of type parameter `T` from `A` to `B`. Although the upper bound `A` is reverted afterward, the AST generated after the inlining phase is not well-typed. But no error is thrown. This case should be studied further.

```
class A
class B extends A

// Original code
def foo[@upper("B") T <: A](x: T) = ???
// After inlining
def foo[@upper("B") T <: B](x: T) = ???
// After several phases
def foo[@upper("B") T <: A](x: T) = ???

foo(new A)
```

Chapter 7

Conclusion

Inspired by the work in Scala 2, we have designed and implemented a macro annotation system that can be used to transform definitions into definitions. We have supported most of the use cases of macro annotation, except for adding definitions to the top-level, and at the same time, have kept the system simple and efficient.

In the design aspect, we have presented a simple but powerful transformation. We also discarded some uncommon used cases, which can be avoided by writing a new macro annotation. In the implementation aspect, we have successfully merged the macro annotation phase into the inlining phase and have implemented a tail-recursive and efficient transformation. By testing different cases, we have shown the functionality of the system.

Bibliography

[1] Eugene Burmako. “Unification of Compile-Time and Runtime Metaprogramming in Scala”. In: (2017), p. 240. DOI: 10.5075/epfl-thesis-7159. URL: http://infoscience.epfl.ch/record/226166.

[2] Shadaj Laddad and Koushik Sen. “ScalaPy: seamless Python interoperability for cross-platform Scala programs”. In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Scala*. 2020, pp. 2–13.

[3] Martin Odersky et al. *Guide to Scala 3 Compiler Contribution*. 2022. URL: https://docs.scala-lang.org/scala3/guides/contribution/arch-phases.html.

[4] Martin Odersky et al. *Scala 3 Reference*. 2022. URL: https://docs.scala-lang.org/scala3/reference/index.html.

[5] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. “A Practical Unification of Multi-Stage Programming and Macros”. In: *SIGPLAN Not.* 53.9 (Nov. 2018), pp. 14–27. ISSN: 0362-1340. DOI: 10.1145/3393934.3278139. URL: https://doi.org/10.1145/3393934.3278139.

[6] Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. “Multi-Stage Programming with Generative and Analytical Macros”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 110–122. ISBN: 9781450391122. DOI: 10.1145/3486609.3487203. URL: https://doi.org/10.1145/3486609.3487203.

[7] Nicolas Stucki, Aggelos Biboudis, Sébastien Doeraene, and Martin Odersky. “Semantics-Preserving Inlining for Metaprogramming”. In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Scala*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 14–24. ISBN: 9781450381772. URL: https://doi.org/10.1145/3426426.3428486.

[8] *Nemerle Wiki Design Pattern*. 2012. URL: https://github.com/rsdn/nemerle/wiki/Design-patterns#Proxy_design_pattern.

[9] *The Rust Reference*. 2022. URL: https://doc.rust-lang.org/reference/index.html.

[10] Dean Wampler. *Programming Scala, 3rd Edition*. O’Reilly Media, Inc., 2021.