Reified Generic Types for Scala 3 on the JVM

by

Jin Xu

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, 2026

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 implements parametric polymorphism on the Java Virtual Machine (JVM) primarily through type erasure. As a result, generic type arguments are normally unavailable during execution, and primitive values passed through generic code are represented using boxed object references. This limits the information available to the virtual machine and prevents generic values from consistently using unboxed forms.

This thesis presents an extension to the Scala 3 compiler that preserves generic type information and communicates it to a reification-aware JVM. The compiler inserts lightweight runtime *reified type values* for method and class type parameters and propagates them through the Scala compiler intermediate representation. It also emits custom JVM classfile attributes that describe how erased method parameters, return values, fields, method invocations, and object allocations relate to the corresponding reified type values. Together, these runtime values and static hints define a reified unboxed semantics, while retaining ordinary JVM bytecode as the underlying executable representation.

The implementation modifies the Scala 3 compiler and integrates reification with its existing transformation and bytecode generation pipeline. It is evaluated using focused generic benchmarks and by recompiling the Scala standard library. The modified compiler successfully compiles the standard library. Of the 2058 standard library tests, 2033 pass, for a pass rate of 98.7%. The remaining failures are primarily caused by structural changes introduced by reification.

These results demonstrate that generic type information can be preserved through the production Scala compiler and encoded at the JVM bytecode level at the scale of a realistic Scala codebase. The work establishes the compiler-side foundation for future evaluation and optimization in a reification-aware JVM.

Acknowledgements

I would like to thank my supervisor, Ondřej Lhoták, who introduced me into Scala and guided me throughout my masters. I am grateful for his patience and guidance throughout the project. Thanks to Gregor Richards and Peter Buhr for reading my thesis and giving valuable suggestions. Thanks to Jian Yu for his help for this project, he makes a wonderful research partner.

I would also like to thank all the members of PLG, you are the ones who made my two-year master's life more enjoyable.

I would like to thank my parents for their love and support throughout the years. They are ever supportive on my academic career. Without their support, I would have never gone down this path.

Finally, I would like to thank my partner Jialin and her cat Xiaoman, for putting up with my daily complaints on compilation errors. Despite not knowing what I was complaining about, they are always there to listen.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scala [14] is a high-level statically typed programming language. Most Scala code is compiled to Java Virtual Machine (JVM) bytecode and executed on the JVM. The JVM [11] is a virtual machine that executes bytecode, a low-level instruction set designed around a set of primitive types, object references, arrays, fields, methods, classes, and interfaces. Targeting the JVM allows Scala to benefit from garbage collection, just-in-time (JIT) compilation, a large standard library, and interoperability with Java code.

However, the JVM is not designed with Scala's advanced type system in mind. Scala has a richer source-level type system than the JVM execution model. For example, Scala supports parametric polymorphism, which allows classes and methods to abstract over types by declaring type parameters [18]. Beyond generic classes and methods, Scala supports features such as type members, higher-kinded types, value classes, and more. The JVM model, on the other hand, operates on bytecode descriptors that distinguish among primitive types, object references, and arrays. Although JVM classfiles can store generic metadata in attributes like the `Signature` attribute, this metadata is not part of the normal bytecode execution semantics [11].

To bridge this gap, Scala, like Java and Kotlin [1], uses a technique called type erasure. Type erasure means that the type information is removed at compile time and replaced with the JVM type model. This allows Scala programs to run on the JVM, but it also means that some of the type information is lost at runtime. As a result, advanced Scala abstractions such as generic classes and generic methods are represented using a smaller set of bytecode level mechanisms.

There are several consequences of this gap: First, generic type information that is explicit in the Scala source program is mostly removed by type erasure, so it is not directly available in the compiled bytecode. Erasure is a common technique used by many other languages that target the JVM; it allows these generic languages to run on the JVM, but it also removes the information that could otherwise be useful to a VM or a just-in-time (JIT) compiler. Second, because erased generic code is represented using object references, primitive values often need to be boxed when passed through generic code and unboxed

when used again in a non-generic context [16]. These boxing and unboxing operations introduce allocation overhead, additional method calls, and less precise information for the JIT compiler. Elimination of such boxing and unboxing is a key goal for this thesis.

One previous work called TASTyTruffle has demonstrated that retaining generic type information at runtime can permit specialized representations and reduce the overhead of erased generics [5]. TASTyTruffle is an interpreter that executes Scala programs directly from their TASTy representation, the "Typed Abstract Syntax Trees" intermediate representation emitted by the Scala 3 compiler. However, interpreting a high-level Scala representation requires implementing the semantics of the supported Scala language constructs in a separate runtime. This makes it difficult to support the full language and existing Scala libraries.

This thesis investigates a different approach. Instead of replacing the Scala compiler or introducing a new high-level interpreter, it modifies the production Scala 3 compiler so that selected generic type information is preserved through compilation and exposed in generated JVM classfiles. The resulting program continues to use JVM bytecode for its control flow, method calls, field accesses, and object allocations. Additional runtime values and classfile metadata describe how erased bytecode values relate to source-level generic types. A reification-aware VM can use this information to execute the bytecode according to a reified unboxed semantics in which the representation of a generic value may depend on its runtime type argument.

The reification system introduced in this thesis communicates generic type information in two complementary forms. First, the compiler inserts explicit reified type values into the program. A reified type value is a lightweight runtime value that identifies a type, such as `Int`, `Boolean`, or a reference type. In the current design, reified type values are represented by JVM `byte` values. Second, the compiler emits static type hints in custom JVM classfile attributes. Passing reified type values through the program is not sufficient by itself because the VM must know which reified type value describes each erased parameter, return value, field, invocation result, and object allocation. The attributes provide this connection.

The emitted classfiles, therefore, contain three related layers:

1. ordinary erased JVM bytecode, which expresses the executable operations;
2. lightweight runtime reified type values, which carry representation-relevant type arguments; and
3. static classfile attributes, which connect erased bytecode values and instructions to the appropriate reified type values.

The combination of these layers defines the input to a reified VM. Such a VM executes the underlying JVM instructions while interpreting generic values according to the additional type information.

The primary scope of this thesis is the compiler side of the system: the design of the reified representation, its implementation in the Scala 3 compiler, and its encoding in JVM classfiles. The execution model assumes a reified VM implemented as an extension of the JVM. A complete implementation and evaluation of VM specialization and performance improvements require the corresponding runtime optimizations and are outside the scope of this thesis. The evaluation instead focuses on whether reification can be integrated into the production Scala compiler deeply enough to support realistic Scala programs.

1.1 Thesis Organization

The remainder of this thesis is organized as follows.

Chapter 2 provides the necessary background on Scala compilation, JVM bytecode, generics, type erasure, primitive boxing, Scala specialization, miniboxing, and TASTyTruffle.

Chapter 3 presents the conceptual design of the reified type hint system. It defines the desired reified unboxed semantics, runtime reified type values, static type hints, layouts for method and class type parameters, captured type parameters, trait reification, and inheritance.

Chapter 4 describes the implementation in the Scala 3 compiler. It introduces the relevant compiler intermediate representations and explains how the compiler phases change the Scala IR to preserve reification information.

Chapter 5 defines the classfile encoding used to communicate reification information to the VM. It describes the class-level, field-level, method-level, and code-level custom attributes.

Chapter 6 evaluates the implementation using custom benchmarks and the Scala standard library. It focuses on language coverage, integration with the production compiler pipeline, and scalability to realistic Scala code.

Chapter 7 compares this work with other approaches to implementing generic types and specialization.

Finally, Chapter 8 summarizes the contributions, limitations, and possible directions for future work.

Chapter 2

Background

This chapter provides the background required for the design and implementation presented in the remainder of the thesis. It first describes how common Scala language constructs are compiled to JVM classfiles. It then explains generics, type erasure, primitive boxing, and existing approaches for retaining or specializing generic type information.

2.1 Compiling Scala Programs to JVM Bytecode

Before discussing generics and type erasure in detail, it is useful to understand how ordinary Scala definitions are represented on the JVM. Scala source programs are compiled into JVM classfiles [17]. The mapping is not one-to-one for every language construct, but many source-level definitions have direct bytecode-level counterparts.

JVM descriptors. In the classfile, there is the notion of JVM descriptors [11]. A JVM descriptor is a compact string stored in a classfile that describes the erased JVM type of a field or method. Field descriptors describe a single JVM type, while method descriptors describe a sequence of parameter types followed by a return type. Primitive types are encoded using one-letter codes, such as `I` for `int`, `Z` for `boolean`, and `V` for `void`. Reference types are written as `L<class name>`.

Scala methods. Scala methods are compiled to JVM methods with JVM descriptors. Method parameters become JVM method parameters, local variables in the method body become JVM local variables, and method bodies are compiled to JVM bytecode instructions.

Scala classes. A Scala class is compiled to a JVM class. Scala class constructors are compiled as JVM constructor methods. The constructor is responsible for invoking the

superclass constructor, initializing the fields, and executing statements that appear in the class body. Class parameters become fields in the generated JVM class. A public `val` parameter is compiled to a private field and a public getter method. A public `var` parameter is compiled to a private field, a public getter method, and a public setter method.

Scala objects. A Scala object is compiled differently from an ordinary class because an object denotes a singleton instance. The Scala compiler emits a JVM class for the object, and it stores the singleton instance in a static field, commonly named `MODULE$`. Methods defined in the object are compiled to methods in the generated class, some of which are exposed through static forwarders.

Scala traits and JVM default methods. Scala traits are compiled mainly to JVM interfaces. Abstract methods of a trait become abstract interface methods. Concrete methods in a trait are compiled to default methods in the generated interface. A default method is an interface method with an implementation body; if a class implements the interface but does not override the method, the implementation in the interface is used.

Since JVM interfaces cannot have fields, fields required by traits are normally implemented in the concrete classes that implement the trait, while the trait exposes abstract getter and setter methods for the fields. Concrete fields in traits are compiled to fields in the implementing class.

Inheritance. Inheritance is also mapped to the JVM class hierarchy. A Scala class can extend at most one class, and this superclass becomes the direct JVM superclass. Traits implemented by the Scala class are compiled as JVM interfaces, and the class implements these interfaces. Method overriding in Scala is therefore implemented using the standard JVM virtual method dispatch mechanism, such as `invokevirtual` for class methods and `invokeinterface` for interface methods.

Scala nestings. Nested, local, and anonymous classes are not stored inside the bytecode of the enclosing class because the JVM classfile format does not support nested classes. Instead, they are hoisted to separate top-level JVM classes with encoded binary names. If such a class needs access to values from its enclosing scope, the compiler represents those values explicitly, by adding synthetic fields and constructor parameters to the generated class. Thus, a source-level nesting relationship in Scala becomes an explicit object-level relationship in JVM bytecode.

Overall, compiling a Scala class to the JVM consists of translating Scala's richer source-level language constructs into the smaller set of abstractions supported by the JVM: classes, interfaces, fields, methods, constructors, descriptors, and bytecode instructions. Features such as trait inheritance, nested classes, and captured variables are made explicit through erasure, synthetic methods, additional classfiles, and classfile metadata.

2.2 Generics and Type Erasure

Generics are a powerful feature in programming languages that allow programmers to write unified code that can operate on different types. Instead of writing a separate implementation for every possible type, a programmer can write one single generic definition parameterized by type variables. Generics provide benefits such as code reusability, compile time type safety, fewer explicit type casting, and more. In real world applications, generics are widely used in data structures, algorithms, and libraries. Data collections are the most common use case of generics.

The JVM is a widely used platform for executing Java and other JVM languages. It executes methods whose parameter and return values are described by JVM descriptors. A descriptor uses JVM type system types such as `I` for `int`, `Z` for `boolean`, array descriptors such as `[I` for `int[]` and reference descriptors such as `Ljava/lang/Object;` for `Object`. On the JVM, primitive values and object references are distinguished by their types. The JVM does not have a notion of generic types, so languages targeting the JVM use type erasure to compile generic code.

For example, consider the following generic Java class. The executable bytecode is conceptually similar to the following erased Java-like representation:[1]

Source code

```
class Pair<K, V> {
    K key;
    V value;
    Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    K getKey() { return key; }
}
Pair<Integer, Integer> IntPair =
    new Pair<>(1, 2);
```

Decompiled Erased Java Bytecode

```
class Pair {
    Object key;
    Object value;
    Pair(Object key, Object value) {
        this.key = key;
        this.value = value;
    }
    Object getKey() { return key; }
}
Pair IntPair = new Pair(
    Integer.valueOf(1), Integer.valueOf(2));

```

In this example, the class `Pair` has two type parameters `K` and `V`. In Java source, the fields `key` and `value` have types `K` and `V`, respectively. However, in the compiled bytecode, due to type erasure, javac replaces the type parameters with their upper bounds, which are `Object` by default. Therefore, in the bytecode, both fields have the descriptor `Ljava/lang/Object;`, which means they are of type `Object`. For the allocation of the `IntPair` instance, the

[1] This example is not literal JVM bytecode. It is Java-like pseudocode used to show the types that appear in the executable JVM descriptors after erasure. There may be future examples of bytecode in this thesis that are also Java-like pseudocode.

information about the type parameters `Integer` is erased. Once on the VM, they are treated as object references.

Scala generics are also source-level abstractions. They are compiled in a similar way. Consider the following Scala class; it is compiled to the following bytecode.

Source code

```
class Box[T](value: T){
    def get: T = value
}
val intBox: Box[Int] = new Box[Int](42)
```

Decompiled Erased Java Bytecode

```
class Box {
    Object value;
    Box(Object value) { this.value = value; }
    Object get() { return value; }
}
Box intBox = new Box(Integer.valueOf(42));
```

This representation makes the program executable on the JVM, but it removes the source-level meaning on generic values. This Scala type erasure is accomplished in a phase of the compilation, called Erasure.

2.3 Primitive Types, References, and Boxing

The JVM instruction set distinguishes between primitive types and object references. For example, an `int` value is loaded with instructions such as `iload` , stored with `istore`, and returned through `ireturn`. An object reference is loaded with `aload`, stored with `astore`, and returned through `areturn`. Similar distinctions exist for fields, method descriptors, and array operations.

This distinction creates a problem for erased generic code. Since erased values are represented as object references, generic code expects object references. However, primitive values such as Scala `Int`, `Double`, etc. are represented on the JVM using primitive values whenever possible. A primitive value cannot be assigned directly to a reference-typed field or passed to a method parameter whose descriptor expects an object reference. Therefore, when a primitive value flows into generic code, it must be boxed into its corresponding wrapper type. When the value later flows back into primitive context, it must be unboxed back to the primitive type.

Consider an explicit variable type in Scala: `val c: Int = id[Int](42)`. The method `id` is a generic method that returns the value passed to it: `def id[T](x: T): T = x`. After Erasure, the generic method `id` and the call are conceptually compiled to the following bytecode:

```
Object id(Object x) { return x; }
Object t1 = Int.box(42);
Object t2 = id(t1);
int c = Int.unbox(t2);
```

The call needs to adapt between the primitive value `42` and the erased generic method type `Object`. In JVM terms, the Scala `Int` must be converted to a boxed instance of `java.lang.Integer` so that it can be passed to the `id` method. After the method returns, the result needs to be unboxed back to an `int` value before it can be used as a Scala `Int`. The Scala compiler now injects these boxing and unboxing operations as runtime helper methods.

The boxing and unboxing is not caused by the programmer explicitly asking for object wrappers. It is a consequence of the erasure-based compilation strategy.

The same issue appears for generic classes. For example, the Scala class `Box` defined above is compiled to a bytecode class with an `Object` field. When creating an instance of `Box[Int]`, the integer value must be boxed to fit into the `Object` field. When at a Scala context where primitive `Int` is expected, the value must be unboxed back to `int`.

From a bytecode instruction point of view, primitive instructions are generally more efficient than object instructions, since they do not have issues like pointer indirection, cache misses, virtual dispatch, and garbage collection overhead.

2.4 Existing Approaches

Several existing approaches address different parts of the problem of generics. This section discusses three approaches that are relevant to this thesis: Scala `ClassTag`, Scala 2 specialization, and TASTyTruffle.

2.4.1 Scala `ClassTag`

The Scala library provides a mechanism called `ClassTag` as a way to carry runtime class information for a statically known type. A value of type `ClassTag[T]` represents the erased runtime class corresponding to the static type `T`. It is commonly used for generic array creation.

The need for `ClassTag` comes from the fact that erased generic code cannot directly create arrays of the type parameter. In the JVM, an array of primitive integers, `int[]`, is represented differently from an array of object references, `Object[]`. If a Scala expression instantiates `new Array[T](length)`, the type parameter `T` is erased. Without additional

information, the compiler cannot know whether it should allocate a primitive array or an array of object references.

If `T` is instantiated with `Int`, the desired representation is an array of primitive integers, `int[]`. If the erased code instead creates an array of object references, `Object[]`, then integers stored in the array must be boxed into `Integer` objects. This loses the representation benefits of primitive arrays.

A common way to write generic array creation in Scala is to require a `ClassTag` using a context bound.

```
def newGenericArray[T: ClassTag](length: Int): Array[T] = new Array[T](length)
```

The syntax `[T: ClassTag]` is called a context bound. A context bound is a shorthand in Scala used to express that an implicit value must exist for a specific type parameter. Such a value is often called a type evidence: it supplies generic code with information or operations that cannot be obtained from the erased type parameter alone. In this case, it means that the method requires an implicit value of type `ClassTag[T]`. In Scala 3, the above definition is equivalent to the following explicit version.

```
def newGenericArray[T](length: Int)(using tag: ClassTag[T]): Array[T] = new Array[T](length)
```

The compiler can then rewrite the generic array allocation so that it uses the available `ClassTag`.

```
def newGenericArray[T](length: Int)(using tag: ClassTag[T]): Array[T] =
    tag.newArray(length)
trait classTag[T] {
  def runtimeClass: jClass[?]
  def newArray(len: Int): Array[T] =
    java.lang.reflect.Array.newInstance(runtimeClass, len).asInstanceOf[Array[T]]
}
```

Thus, a call such as `newGenericArray[Int](10)` is compiled as a call that passes an available `ClassTag[Int]` instance. At runtime, this tag contains enough information to allocate an array with the appropriate JVM representation.

`ClassTag` is a useful mechanism for certain use cases, but it has several limitations. First, it must be explicitly requested by the programmer. If a method does not have a

`ClassTag` parameter, then it cannot access the runtime class information. Second, `ClassTag` can only represent the erased runtime class, not the actual Scala type. It can distinguish between a Scala `Int` versus `String`, but it cannot distinguish between a `List[Int]` and a `List[String]`, since both are erased to the same runtime class. Third, `ClassTag` is not a general mechanism for optimizing generic code. It is a library-level mechanism, and it helps user programs to perform certain runtime operations. Within the Scala compiler, it is primarily used for generic array creation, and it does not provide a way to eliminate boxing and unboxing in generic code.

2.4.2 Scala 2 Specialization

Scala 2 allows specialization through the `@specialized` annotation [4]. Specialization generates separate versions of generic code for primitive types, which can avoid boxing and unboxing inside the specialized code. The programmer can control what types the compiler specializes on by giving the annotation specific types.

When a type parameter is annotated with `@specialized`, the Scala compiler fully specializes the code for all primitive types. For example,

```
1 class Box[@specialized T](t: T):
2     def get: T = t
```

The Scala 2 compiler generates versions of the `Box` class: `Box$mcI$sp` for `Int`, `Box$mcD$sp` for `Double`, and so on for all primitive types, as well as a generic version `Box`. Each specialized version uses the primitive type directly, so there is no boxing or unboxing when using the specialized versions.

A downside of full specialization is code bloat. For the specialized `Box` class, the compiler generates 10 versions of the class: one for each of the 9 primitive types and one for reference types. The number increases exponentially with the number of specialized type parameters. As a result, the compiled classfile sizes can increase significantly.

Scala 2 did offer a way to control the specialization by giving specific types in the annotation [4]. For example, `class Box[@specialized(Int, Double) T]` would only specialize the class for `Int` and `Double`. There is also the `@unspecialized` annotation to exclude specific methods from specialization. This can be of use when the programmer wants to explicitly avoid generating specialized code for certain methods. These mechanisms are not available as a supported feature in Scala 3.

Generating classes for specialization adds complexity to the class hierarchy. The generic class remains as the common supertype, and the specialized classes are subclasses of the generic class. As shown in figure 2.1, the specialized classes `Box$mcI$sp`, `Box$mcD$sp`, etc. are subclasses of the generic class `Box`.

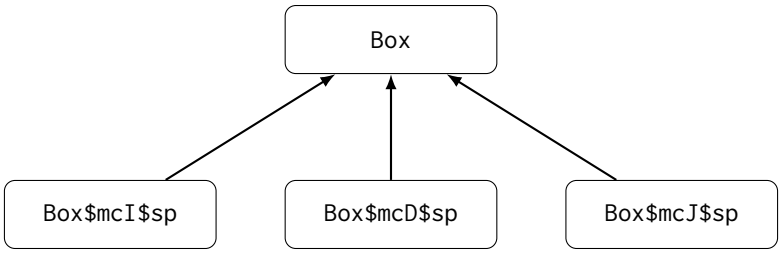

Figure 2.1: Class hierarchy produced by Scala 2 specialization for `class Box[@specialized T]`.

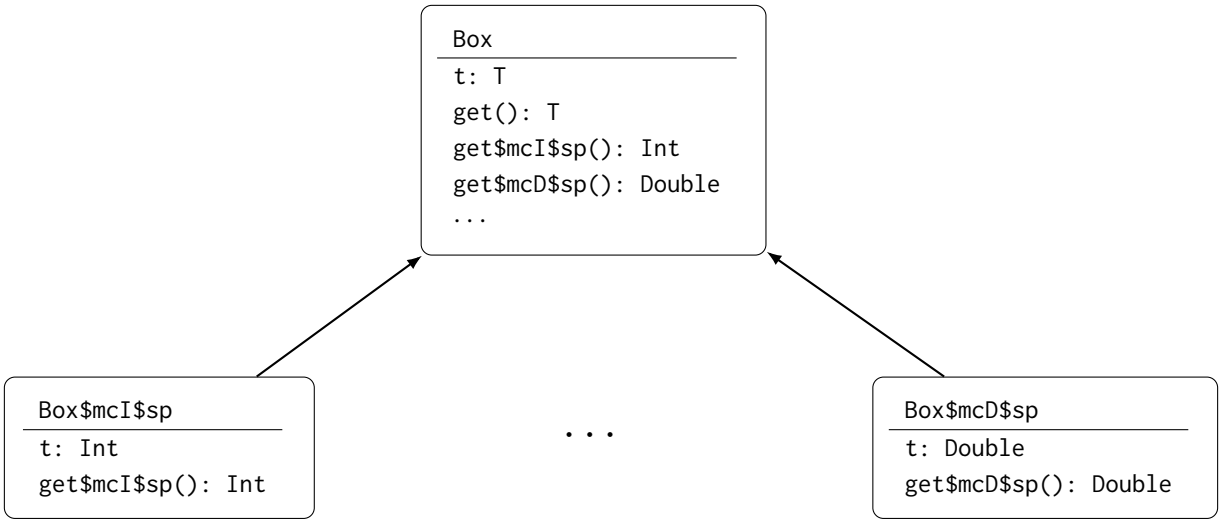

Figure 2.2: Simplified class hierarchy for `class Box[@specialized T](t : T){def get: T}`.

Having fields and methods further increases the complexity of specialization. For example, class `Box` defines the method `get`. So when the compiler generates specialized code, it not only generates the subclasses, but it also generates specialized versions of the method `get` in all subclasses. As shown in figure 2.2, the generic class remains the common supertype, so it must provide a generic method `get`, as well as specialized entry points for all specialized types. Each specialized subclass then overrides the specialized method corresponding to its primitive type. For example, the specialized class `Box$mcI$sp` for `Int` overrides the method `get$mcI$sp()` that returns an `Int`. These forwarding and overriding methods preserve compatibility between generic and specialized call sites, but they also increase the number of generated methods.

Overall, specialization is a powerful technique that can generate the most efficient code for primitive types since it requires no boxing or unboxing in the specialized versions. However, it has tradeoffs. First, it can cause code bloat, and the number of generated variants grows exponentially in the number of specialized type parameters. Second, it requires the

programmer to explicitly annotate the code, so for code that is not annotated, for example, old generic libraries, it does not provide any performance benefits. Third, boxing and unboxing are still needed at the boundary between specialized code and ordinary erased code.

2.4.3 TASTyTruffle

TASTyTruffle is another approach to recover the performance loss of erasure. Instead of executing ordinary JVM bytecode, TASTyTruffle interprets the TASTy. Unlike JVM bytecode, where generic type information has mostly been erased, TASTy is an easily parseable representation of the original Scala source code. TASTy preserves the full Scala type information, including generic types. By executing TASTy directly, TASTyTruffle can access the full type information at runtime, and it reifies generic types directly inside the interpreter and uses this information during execution [5].

This approach has an important advantage: it starts from a representation that has not lost as much type information as bytecode. Therefore, it can potentially reason about Scala programs at a higher semantic level than bytecode.

TASTyTruffle is implemented using the Truffle framework, which is a framework for building language interpreters that can be optimized by the Graal just-in-time (JIT) compiler. By using Truffle, TASTyTruffle can benefit from the optimizations provided by the Graal JIT compiler, such as inlining, partial evaluation, and escape analysis. This allows TASTyTruffle to achieve good performance even though it is an interpreter.

The central idea of TASTyTruffle is that generic type information should guide data representation and code generation. Since TASTy preserves the full Scala type information, the interpreter can pass reified type objects into generic code. The reified types are then used to choose more precise runtime representations. For example, a generic value whose type argument is known to be `Int` can be represented using a primitive `int` value rather than a boxed `Integer` object. Similarly, arrays of primitive types can be represented using primitive arrays rather than arrays of boxed objects. By using the full type information, TASTyTruffle can avoid boxing and unboxing operations that would otherwise be present in erased normal bytecode.

TASTyTruffle also performs dynamic specialization of generic code. Rather than generating all specialized variants ahead of time, it specializes code at runtime based on the reified value of types during execution. This differs from traditional specialization like Scala 2 specialization, which duplicates code statically during compilation. Dynamic specialization also allows the JIT to perform aggressive optimizations based on the actual types observed at runtime, which leads to more efficient specialized code.

However, TASTyTruffle also has limitations. It is a research prototype that implements only a core subset of Scala rather than the full language. The paper states that it supports primitives, classes with single inheritance, singleton objects, conditionals, loops, etc. More

complicated features such as traits, higher-kinded types, and Java interoperability are outside the scope of the project. This limits its ability to run realistic Scala programs and the full Scala standard library without substantial additional implementation effort. Also, since it is an interpreter, it may not achieve the same level of performance as compiled code, even with JIT optimizations.

Chapter 3

Design of a Reified Type Hint System

The previous chapter described how Scala programs are compiled to JVM bytecode and how generic type information is erased before code generation. For example, the source-level expression `new Box[Int](42)` states that the type argument for `Box` is `Int`. However, after the Erasure phase, the generated bytecode does not directly expose this information to the JVM. The class `Box` is allocated for an erased type parameter, and fields or methods involving that type parameter are represented using the erased JVM type `Object`. This chapter presents the design of a reification system that preserves selected generic type information beyond the Erasure phase.

The goal of this thesis is to make it possible to execute such erased JVM bytecode with a reified, non-standard semantics. Under ordinary JVM semantics, generic primitive values are represented as boxed objects when they flow through erased generic code. The desired semantics instead allows a reified VM to represent such values using primitive representations when the runtime type information indicates that they are primitive. To make this possible, the compiler augments ordinary JVM bytecode with enough additional information for the reified VM. The reified VM then recovers the representation intended by the source-level generic types, and executes the bytecode at runtime with the desired semantics.

This is done in two complementary ways. First, the compiler inserts runtime reified type values into the Scala IR. These are ordinary runtime values of compiler-internal type `ReifiedValue`, represented as a JVM `byte`, that distinguish cases such as `Int`, `Boolean`, a reference value, or a type parameter introduced by a method or class. Second, the compiler emits bytecode attributes that describe how erased bytecode values relate to source-level generic types. These attributes, which serve as type hints, are then read by a reification-aware runtime, referred to in this thesis as a reified VM.

The system does not replace Scala's erased compilation strategy. Instead, it adds an auxiliary reification alongside the erased JVM program. A standard JVM can still execute the bytecode according to ordinary erased semantics, while a reified VM can inspect the additional hints to recover information that would otherwise be unavailable.

This chapter describes the design at a conceptual level. The compiler transformation implementations are described in Chapter 4, and the bytecode attributes are described in Chapter 5.

3.1 Motivation: From Erased to Source-Level Semantics

The desired semantics is a reified unboxed semantics. In that semantics, a generic bytecode value can have a representation determined by a runtime type argument. If the runtime type argument is `IntTag`, the value is then represented as a primitive `int`; if it is `ReferenceTag`, the value is carried as an object reference. Reified type values provide the runtime information needed to choose the representation, while bytecode type hints tell the VM which reified type value should be used to interpret each erased value.

The reified unboxed semantics uses runtime type values such as `IntTag`, `BooleanTag`, and `ReferenceTag`. These values are called reified type values in this thesis. Their main role is to give the reified VM enough runtime information to choose an appropriate representation for generic values and to reduce unnecessary boxing and unboxing when a type argument is known to be primitive. The reified VM can also use reified type values for optimizations such as aggressive speculation, inlining, and partial evaluation. This can lead to efficient code that is close to hand-written code for specific type arguments. However, the implementation of the reified VM is out of scope for this thesis. The main focus of this thesis is the design and implementation of the reification system that encodes enough information to enable the reified unboxed semantics.

In the transformed Scala program, most ordinary operations do not inspect the reified type values directly. For example, the program does not branch on `IntTag` vs `BooleanTag`. The main direct source-level use of a reified type value is generic array construction. In ordinary Scala, constructing an array whose element type is a type parameter requires a `ClassTag`, a runtime type evidence, and is achieved through Java reflection. Having the reified type values available allows the compiler to generate code that constructs arrays without reflection.

3.1.1 Generic Method Calls

Consider a simple generic method:

Source code

```
def id[T](x: T): T = x
val c: Int = id[Int](42)
```

Decompiled Erased Java Bytecode

```
Object id(Object x) = { return x;}
Object t1 = Int.box(42);
Object t2 = id(t1);
int c = Int.unbox(t2);
```

At a Scala source-level, this program means that method `id` takes a value of type `T` and returns a value of the same type `T`. The call `id[Int](42)` explicitly instantiates `T` with `Int`. Therefore, the argument `42` is a primitive `Int` value, and the result of the call is also a primitive `Int` value. The assignment to `c` does not require any conceptual change of representation since `c` is also of type `Int`.

However, this is not the representation used by ordinary JVM bytecode. Scala compiles to the JVM, and JVM generics are erased. After the Erasure phase, the method type parameter `T` is no longer directly represented in the method descriptor. Since `T` may stand for a reference type, the erased representation of `T` is generally `java.lang.Object`.

In the erased representation, the source-level relationship between the parameter type and the return type is lost in the JVM descriptor. The method no longer expresses that the return type is the same as the parameter type, which is type `T`. It only expresses that the parameter and return type are `Object`.

This loss of information has an important impact on primitive values. The value `42` is a primitive `Int` at the Scala source level, but the erased method expects an `Object`. Therefore, the compiler must box the primitive value `42` into a `java.lang.Integer` object before passing it to the method. The method `id` then returns an `Object`, which must be unboxed back to an `Int` before it can be assigned to `c`. This boxing and unboxing is a direct consequence of Erasure, and it can have a performance cost.

This behavior is needed for correctness for a standard JVM, but it is not the semantic model that the Scala source program expresses. The Scala program does not intend to forget the fact that `T` is instantiated with `Int`, and it does not intend to treat the argument and result as `Object` values. The erased JVM bytecode forgets this because the JVM does not have a way to represent it.

Thus, the design in this thesis is motivated by a different abstract target semantics. Instead of thinking of generic values as always erased to `Object`, one can imagine a reified target language, called the reified unboxed language, where generic methods receive run-time type arguments. In the reified unboxed language, the example above would look like this:

```
A(T) id(T: Type, x: A(T)){ return x; }
int c = id(IntTag, 42);
```

Here, `T` is a runtime type value. For example, `T` may be `IntTag`, `BooleanTag`, or `ReferenceTag`. The notation `A(T)` is a dependent type: the representation of the value depends on the runtime type value `T`. That is, `A` is a type-level function that maps a runtime type value to a value type.

```
A(IntTag)       =  Int
A(BooleanTag)   =  Boolean
A(ReferenceTag) =  Object
...             =  ...
```

So now, the type of parameter `x` is not simply `Object`, but it is `A(T)`, meaning "the value representation determined by current type argument `T`".

For the call `id[Int](42)`, the runtime type argument is `IntTag`, so:

```
T    =  IntTag
A(T) =  A(IntTag) = Int
```

Therefore, the call behaves like `id(IntTag, 42)` where the argument and return value are both of type `Int`, as the source-level program expresses. No conceptual boxing or unboxing operations are needed in the desired reified semantics.

3.1.2 Generic Class Allocation

The same issue appears for generic class allocation, not only for generic method calls. Consider:

Source code

```
class Box[T](value: T){
  def get: T = value
}
val b: Box[Int] = new Box[Int](42)
val c: Int = b.get
```

Decompiled Erased Java Bytecode

```
class Box {
  Object value;
  Box(Object value) { this.value = value; }
  Object get() { return value; }
}
Object t1 = Int.box(42);
Box b = new Box(t1);
Object t2 = b.get();
int c = Int.unbox(t2);
```

At the Scala source level, the allocation `new Box[Int](42)` constructs a `Box` object whose type parameter is `Int`. The method `get` returns a value of type `Int`.

Under ordinary JVM erasure, the type parameter `T` of `Box` is erased. The type argument `Int` is not directly represented in the JVM bytecode of the allocated object. The compiled class behaves as if the field and method use `Object`.

This is correct for a standard JVM, but it does not reflect the source-level semantics. The source program expresses that the allocated object is a `Box[Int]`, and that the method `get` returns an `Int`. For our desired reified unboxed semantics, the desired object layout is different. The object should carry the runtime type argument for `T`; when that argument is `IntTag`, the language semantics should treat the field `value` and the return type of method `get` as primitive `Int` rather than a boxed `Integer`.

In the reified unboxed language, the allocation would look like this:

```
class Box{
  T: Type;
  A(T) value;
  Box(T: Type, A(T) value) { this.T = T; this.value = value; }
  A(T) get() { return value; }
}
Box b = new Box(IntTag, 42);
int c = b.get();
```

Here, the type of the field is not fixed as `Object`, but it is `A(T)`, where `T` is a runtime type argument stored in the object instance. If the object is created with `T = IntTag`, then `A(T) = A(IntTag) = Int`. Thus, the field `value` and the return type of method `get` are conceptually primitive integers. This is the source-level semantics that the Scala program expresses, and it is the semantics that the reification design in this thesis aims to recover.

The two examples show the same generic principle:

The representation of a generic value (parameter, return value, or field) depends on its type argument.

This is a key idea behind the reified unboxed language. Instead of treating every erased generic value as an `Object`, the reified unboxed language treats generic values as having a dependent type `A(T)`. The runtime type value `T` determines whether a value is represented as a primitive type or a reference type.

3.1.3 `A(T)` as a Dependent Type

The notation `A(T)` is a dependent type in the reified unboxed language. It is dependent because the type of the value depends on another runtime value, in this case the runtime type argument `T`.

In ordinary JVM bytecode, the type of a local variable or a method parameter is fixed by the JVM descriptor. For example, the parameter `x` of method `id` has type `Object` in the JVM descriptor. But in the reified unboxed language, the parameter type may depend on a runtime type tag: `x: A(T)`. This means that the representation of `x` cannot be determined only from the method descriptor. Instead, it depends on the value of `T` at runtime. For example:

```
if T = IntTag       :  x: A(T) = Int
if T = BooleanTag   :  x: A(T) = Boolean
if T = ReferenceTag :  x: A(T) = Object
```

3.1.4 Value Compatibility

Standard JVM semantics distinguishes primitive representations, such as `I` for `Int`, from reference representations, such as `L` for object references. It has no representation for a location whose actual contents may be either primitive or reference, depending on a runtime type argument. The reified unboxed semantics adds such a representation, called `ObjectAny`.

Table 3.1 shows the compatibility rule for `Int` and `Integer`; the same rules apply to other primitive types and their boxed counterparts. A compatibility rule specifies which runtime values are legal inhabitants of a particular bytecode-level or reified type value representation. The standard JVM has only the `I` and `L` columns. The reified unboxed semantics adds `ObjectAny`.

Runtime value	Type representation		
	`I`	`L`	`ObjectAny`
primitive `int`	yes	no	yes
boxed `Integer`	no	yes	yes
ordinary reference	no	yes	yes

Table 3.1: Compatibility between runtime values and type representations.

For type `I`, only primitive `int` values are compatible.

For type `L`, it is an ordinary reference type. Boxed primitives and ordinary references are compatible, since both are object references. However, primitive values are not compatible with `L` since they are not object references.

For the new type `ObjectAny`, all three kinds of values are compatible. Since it represents a value whose representation is determined by a type argument, it may correspond to a primitive type, a boxed type, or an ordinary reference type, depending on the value of the type argument. Therefore, all three kinds of values are compatible with `ObjectAny`.

This is a key motivation for having a new type representation `ObjectAny` in the reified unboxed language. After erasure, a generic value is always represented as `Object`, making

primitive values incompatible with the erased type. In the reified unboxed language, the new type `ObjectAny` allows primitive values to be compatible with generic types, as long as the runtime type argument indicates that the value is a primitive type.

3.1.5 Encoding the Modified Semantics on Top of JVM Bytecode

The desired reified unboxed language is not ordinary JVM bytecode. It has a different type system and different value compatibility rules. It contains dependent types such as `A(T)`, and object representations that depend on runtime type arguments. This means that the reified unboxed language cannot be directly represented in ordinary JVM bytecode.

So why not design a new bytecode format that directly expresses the reified semantics?

The answer is that the desired language is close enough to JVM bytecode that we can encode it using ordinary bytecode plus additional information. The erased JVM program already contains most of the control flow, method calls, object allocations, and field accesses that the reified unboxed language would have. What is missing is the runtime type information and the connection between erased values and source-level generic meanings.

Therefore, the modified Scala compiler does not generate a completely new instruction set. Instead, it generates JVM bytecode with two additions:

- explicit reified type values
- metadata that describes how erased bytecode values relate to source-level generic types

There is also the concern of implementing the reified VM. If the reified semantics were expressed in a completely new bytecode format, then the reified VM would have to implement a new instruction set and a new type system. By contrast, the design in this thesis allows the reified VM to reuse the existing JVM instruction set and type system, while only adding support for interpreting the additional metadata and reified type values. This makes the implementation of the reified VM more feasible. In fact, the reified VM is implemented as a fork of the Espresso JVM, which is a JVM implementation in Java [6].

So, the overall workflow is:

Ordinary Scala compilation

$$\texttt{Scala source} \xrightarrow{\texttt{scalac}} \texttt{erased JVM bytecode} \longrightarrow \texttt{standard JVM(e.g. Espresso)}$$

Reified compilation

$$\texttt{Scala source} \xrightarrow{\texttt{modified scalac}} \texttt{reified unboxed language (JVM bytecode + metadata)}$$
$$\longrightarrow \texttt{Interpreter for reified unboxed language (reified VM, forked Espresso)}$$

The reified pipeline still uses JVM bytecode as the low level representation, but its intended semantics is the reified unboxed language. The above is the central motivation for the design of the reification system in this thesis. The design is guided by the following goals, which are described in the next section.

3.2 Design Goals

The design of the reified type values and the bytecode annotations is guided by five main goals.

Preserve Useful Generic Type Information After Erasure. At the source level, a generic call `id[Int](42)` explicitly instantiates the type parameter with `Int`. After the Erasure phase, however, the type parameter is no longer visible. The method parameter and return type are represented using erased JVM types. The fact that `T = Int` is therefore no longer directly visible in bytecode.

The reification system preserves this information by creating a runtime value for the type argument and by recording type hints that associate the erased parameter and return value with the original method type parameter. Thus, although the bytecode remains erased, the generated metadata and extra runtime value still describe the source-level meaning.

Avoid Full Specialization. One possible way of preserving type information would be to generate specialized versions of generic methods and classes. For example, the compiler could generate separate versions of `id` for `Int`, `Boolean`, and reference types. This can produce efficient code, but code size grows quickly with the number of type parameters.

The reification approach keeps generic methods and classes in erased form, alongside lightweight runtime values and type hints. This avoids eager duplication of code for every type argument and therefore avoids code explosion.

Lightweight Runtime Representation and Static Type Hints. The design separates two related but different notions. The first is the set of reified type values that the program carries at runtime. The second is the set of static hint descriptors stored in metadata.

The reified type values are intended to preserve useful information without introducing heavy runtime burden. In particular, a reified type value does not attempt to encode all Scala types. Scala has a rich type system, and a complete runtime encoding for it would be far more expensive in both compiler complexity and runtime overhead. Instead, the system uses a lightweight runtime representation, `ReifiedValue`, which identifies base representation categories, such as primitive types and reference types.

In this implementation, `ReifiedValue` is represented as a JVM `byte`. The byte can be passed as an extra method argument or stored as a field. This representation is intentionally compact, since reified type values are inserted into many generic methods and classes.

Static type hints may refer to the base primitive categories, but they may also say that the relevant runtime reified type value must be obtained from a method type parameter, a class field, or a trait accessor.

This separation keeps the runtime representation lightweight, while still allowing the metadata to describe where the reified VM should look for the relevant runtime type argument.

Support Method and Class Type Parameters. Method type parameters and class type parameters have different lifetimes. A method type parameter exists for a single method invocation. Thus, reified type values for method type parameters are passed as additional method arguments.

Class type parameters, however, belong to the object instance. Any class type argument must remain available after the execution of the constructor, since instance methods may need to refer to it later. Therefore, class type parameters require object-level storage. A design requirement is to support both cases in a uniform way.

Expose Information to the Reified VM. The final goal is to communicate type information to a reified VM. Passing reified type values through the program alone is not enough. From the runtime's point of view, it sees bytecode instructions, local variables, fields, method descriptors, and classfile structures. It does not observe the Scala IR directly. Therefore, the modified Scala compiler must also emit metadata that tells the runtime how reified type values correspond to bytecode-level constructs.

For example, at a bytecode allocation site for any generic class, the runtime sees a `NEW` instruction that allocates an erased JVM class. The metadata must tell the runtime which local variables contain the reified type arguments for that allocation. Similarly, for a method call whose erased return type is `Object`, the metadata must describe the source-level meaning of that invocation result. For boxing and unboxing calls that were necessary for erased bytecode semantics but unnecessary for the reified unboxed semantics, the metadata also marks them as unnecessary.

3.3 Running Example

Consider the following example.

The program contains both a generic class and a generic method, along with allocations and invocations of the class and method. The class `Box[A]` stores a value of type `A`, and

```scala
class Box[A](val value: A) {
  def get: A = value
}

def makeBox[T](x: T): Box[T] =
  new Box[T](x)

val b: Box[Int] = makeBox[Int](42)
val v: Int = b.get
```

the method `makeBox[T]` constructs a `Box[T]` from a value of type `T`. At the final call site, `T` is instantiated with `Int`, so the source program clearly contains the information that the result has type `Box[Int]`.

Under normal JVM erasure, the type parameter `A` of `Box` and the type parameter `T` of `makeBox` are not directly represented as runtime type arguments. The field `value` is compiled using an erased representation, `java.lang.Object`, and the allocation of `Box[Int]` does not expose the concrete type argument to the JVM.

Conceptually, the reification design transforms the Scala program into the following form. Throughout this thesis, all examples of reified Scala represent the IR right before the backend phase.

Standard erased code before backend

```scala
class Box(value: Object) {
  def get: Object = value
}

def makeBox(x: Object): Box =
    new Box(x)
val b: Box =
    makeBox(Int.box(42))
val v: Int = Int.unbox(b.get)
```

Reified compilation before backend

```scala
class Box(reifiedField$Box$A: ReifiedValue, value:
↪  Object) {
  val reifiedField$Box$A: ReifiedValue =
  ↪  reifiedField$Box$A
  def get: Object = value
}
def makeBox(reified$makeBox$T: ReifiedValue)(x: Object):
↪  Box =
  val reifiedLocal$Box$A = reified$makeBox$T
  new Box(reifiedLocal$Box$A, x)
val b: Box = makeBox(IntTag)(Int.box(42))
val v: Int = Int.unbox(b.get)
```

The method `makeBox` is generic, so it takes an extra parameter list that represents its type parameter – in this case, `reified$makeBox$T`. The constructor of `Box` is also transformed to take an additional argument (`reifiedField$Box$A`) whose runtime value represents the class type parameter `A`. The additional argument is stored in a field of the class, so that it remains available for the lifetime of the object.

The actual type argument for `T` at the call to `makeBox`, `Int` type, is represented by a reified type value, here written as `IntTag`. Inside `makeBox`, because it constructs a `Box[T]`, the reified type value for method type parameter `T` is copied into a local representing the reified argument for type `Box.A`. That reified argument is then passed to the constructor of `Box`, where it is stored as the runtime representation of the class type parameter `A`.

The compiler also records static type hints for this example. Informally, the type hints say:

in class `Box` `K(0)` corresponds to	:	field `reifiedField$Box$A`
`Box.A`	:	class type parameter `K(0)` in `Box` layout
field `Box.value`	:	`K(0)`
method `Box.get` return	:	`K(0)`
in method `makeBox` `M(0)` corresponds to	:	parameter `reified$makeBox$T`
`makeBox.T`	:	method type parameter `M(0)` in `makeBox` layout
parameter `makeBox.x`	:	`M(0)`
return of `makeBox`	:	a reference
allocation `new Box[T]`	:	constructor reified argument comes from `M(0)`
boxing on 42	:	unnecessary
`b.get` return	:	primitive `Int`
unboxing on `b.get`	:	unnecessary

Later sections define the runtime reified type values, the method and class layouts, and the bytecode-level hint descriptors more precisely.

3.4 Type Hints and Reified Type Values

The compiler uses two related but distinct representations of type information. The first representation is a type hint, which is a static compiler-level description of the source-level semantics. Type hints are used by the compiler and backend to record information in classfile attributes. The second representation is a reified type value, which is a compact runtime value inserted into the generated program. Reified type values are passed as additional method arguments, stored in objects, or accessed through generated accessor methods.

In the implementation, the hints in the static type-hint model use an enumeration called `TypeB` to describe the source-level Scala types.

```
TypeHint = Primitive(p)
         | Reference
         | M(i)
```

```
| K(i)
| Array(TypeHint)
| None
| Receiver
```

Each hint form records a different relationship between an erased JVM value and the source-level Scala type from which it is derived. Table 3.2 summarizes the meaning of each form.

Hint	Meaning
`Primitive(p)`	a known primitive type such as `Int` or `Boolean`
`Reference`	an ordinary object-like value
`M(i)`	the i-th method type parameter
`K(i)`	the i-th class or trait type parameter
`Array(t)`	an array whose element type has hint `t`
`None`	no useful reified hint is available
`Receiver`	parameter for which the VM finds the `K(n)` value

Table 3.2: Conceptual type-hint language.

A reified type value is a compact runtime representation of selected type information. The design does not attempt to encode all Scala types. Instead, it only represents the categories needed by the reified VM to execute the reified unboxed semantics. These categories include primitive types, object-like types, method type parameters, and class or trait type parameters. Table 3.3 shows the runtime representation.

Source-level type information	Reified representation
`Byte`	`ByteTag` (`B`)
`Char`	`CharTag` (`C`)
`Double`	`DoubleTag` (`D`)
`Float`	`FloatTag` (`F`)
`Int`	`IntTag` (`I`)
`Long`	`LongTag` (`J`)
`Short`	`ShortTag` (`S`)
`Boolean`	`BooleanTag` (`Z`)
Object-like type	`ReferenceTag` (`L`)
Method type parameter `T`	reference in the Scala IR to the reified method parameter
Class type parameter `A`	reference in the Scala IR to the reified field or accessor method

Table 3.3: Runtime representation of reified type values.

The system does not encode all Scala types. For example, refinement types and higher-kinded types are treated as object-like values in this representation. A complete runtime encoding of Scala types would be significantly heavier and would require substantially more compiler and runtime support.

Using this compact representation has several advantages. Reified type values can be passed as ordinary method arguments, stored in objects with limited layout overhead, and inspected directly by the reified VM. Type hints, meanwhile, provide a stable static description that survives the Erasure phase and can be injected into classfiles as metadata. Chapter 5 describes the concrete classfile attributes that encode these hints.

3.5 Reifying Method Type Parameters

Method type parameters are reified by passing additional runtime values to generic methods. Consider the following method.

Source code

```
def choose[A, B](x: A, y: B): A = x
val c: Int = choose[Int, String](42,
↪  "one")
```

Erased code before backend

```
def choose(x: Object, y: Object): Object = x
val c: Int = Int.unbox(choose(Int.box(42),
↪  "one"))
```

At each call to `choose`, the caller knows the type arguments for `A` and `B`. Therefore, the compiler can generate reified type values at the call site and pass them to the method. The transformed method has the following conceptual form.

```
def choose(reified$choose$A: ReifiedValue, reified$choose$B: ReifiedValue)
    (x: Object, y: Object): Object = x
val c: Int = Int.unbox(choose(IntTag, ReferenceTag)(Int.box(42), "one"))
```

A call to `choose[Int, String]` with arguments `1` and `"one"` is transformed by inserting the reified-value argument list: `choose(IntTag, ReferenceTag)(1, "one")`.

In this example, `Int` is a primitive type, so it receives the tag `IntTag`; `String` is a reference type, so it receives the reference tag.

To describe method type parameters after erasure, the design assigns each method type parameter a method index. For example, the method `choose` introduces type parameters `A` and `B`. These are represented as `M(0)` and `M(1)` respectively. The notation `M(i)` means the i-th type parameter introduced, and possibly captured, by the current method.

For the `choose` example, the desired reified unboxed semantics can be described by the following type hints.

in method `choose` hint `M(0)` corresponds to	:	parameter `reified$choose$A`
in method `choose` hint `M(1)` corresponds to	:	parameter `reified$choose$B`
method `choose` parameter `x` type	:	`M(0)`
method `choose` parameter `y` type	:	`M(1)`
method `choose` return type	:	`M(0)`
box on 42	:	unnecessary
method `choose` return	:	primitive `Int`
unbox on `choose` return	:	unnecessary

After erasure, this information is not visible in the JVM descriptor, but the hint representation still records it. The backend can therefore emit metadata showing that erased parameters and results are not merely `java.lang.Object` values, but values whose source-level meaning depends on method type parameters. The backend also emits metadata that marks unnecessary boxing and unboxing calls. Chapter 5 explains the metadata in more detail; this chapter simply states that this reified transformation makes the backend able to describe the desired source-level semantics that the reified unboxed language aims to achieve.

Nested methods and closures may also need reified type values from enclosing method invocations. Those cases require capture rules because the nested method has its own method-local index space. Section 3.8 describes the capture layouts separately.

3.6 Reifying Class Type Parameters

Class type parameters are different from method type parameters because their reified type values must remain available for the lifetime of an object. A method type parameter can be passed as an additional argument at each method call. A class type parameter, however, is part of an object instance. Therefore, once an object is constructed, its methods must still be able to recover the reified type values corresponding to the class type parameters.

Consider the following generic class.

Source code

```
class Pair[A, B](fst: A, snd: B) {
  def first: A = fst
  def second: B = snd
}
val p: Pair[Int, String] =
    new Pair[Int, String](42, "hello")
val p1: Int = p.first
```

Erased code before backend

```
class Pair(fst: Object, snd: Object) {
  def first: Object = fst
  def second: Object = snd
}
val p: Pair =
    new Pair(Int.box(42), "hello")
val p1: Int = Int.unbox(p.first)
```

The class `Pair` introduces two type parameters, `A` and `B`. A value of type `Pair[Int, String]` must remember that its first component has type `Int` and its second component has type `String`. This information cannot be passed only to individual method calls, because the methods `first` and `second` may be called long after the object has been constructed – making it hard to recover the relevant type information at method call time.

Therefore, the compiler changes the object layout of `Pair` to include additional fields that store the reified type values for the class type parameters. The constructor also receives one reified type value for each class type parameter, and stores the values in compiler-generated fields.

Conceptually, the transformed class has the following form.

```
class Pair(reifiedField$Pair$A: ReifiedValue, reifiedField$Pair$B: ReifiedValue)(fst: Object,
↪  snd: Object) {
  val reifiedField$Pair$A: ReifiedValue = reifiedField$Pair$A
  val reifiedField$Pair$B: ReifiedValue = reifiedField$Pair$B
  def first: Object = fst
  def second: Object = snd
}
val reifiedLocal$Pair$A: ReifiedValue = IntTag
val reifiedLocal$Pair$B: ReifiedValue = ReferenceTag
val p: Pair =
    new Pair(reifiedLocal$Pair$A, reifiedLocal$Pair$B)(Int.box(42), "hello")
val p1: Int = Int.unbox(p.first)
```

The fields `reifiedField$Pair$A` and `reifiedField$Pair$B` are not part of the original program. They are now part of the object state, and are inserted by the compiler to preserve information about the class type parameters `A` and `B`. Methods such as `first` and `second` can then refer to the stored reified type values through the class-level type hint model.

At the allocation site, the compiler knows that `A` is instantiated with `Int` and `B` is instantiated with `String`. It therefore generates reified type values for the two type arguments and passes them to the constructor.

To describe class type parameters within the class after the Erasure phase, the design assigns each class-related type parameter a class index. The notation `K(i)` means the i-th type parameter introduced by the current class or trait. For the `Pair[A, B]` example, the source-level signature can be described by the following hints.

```
A  :  K(0)
B  :  K(1)
```

Using this mapping, the the desired reified unboxed semantics can be described by the following hints:

in class `Pair` hint `K(0)` corresponds to	:	field `reifiedField$Pair$A`
in class `Pair` hint `K(1)` corresponds to	:	field `reifiedField$Pair$B`
field `fst` type	:	`K(0)`
field `snd` type	:	`K(1)`
method `first` return type	:	`K(0)`
method `second` return type	:	`K(1)`
local `reifiedLocal$Pair$A`	:	reified type value for `Pair.A`
local `reifiedLocal$Pair$B`	:	reified type value for `Pair.B`
boxing for 42	:	unnecessary
method (p.first) return	:	primitive `Int`
unboxing for `p.first`	:	unnecessary

After the Erasure phase, the JVM descriptor of `first` and `second` does not show that their results have source-level types `A` or `B`. However, the hint representation still records these relationships. To bridge the type system gap introduced by type erasure, the Scala compiler inserts boxing and unboxing operations. These operations are unnecessary for the desired reified unboxed semantics. The backend can therefore emit metadata indicating that the return types of `first` and `second` depend on the class reified type values `K(0)` and `K(1)`, and that there exist unnecessary operations in the bytecode. Metadata details are described in Chapter 5.

Nested classes and local classes may also need reified type values from enclosing classes or methods. Those cases require captured class-level layout because the nested object must carry all the reified type values needed by its methods after construction. Section 3.8 describes the capture layouts separately.

3.7 Reifying Trait Type Parameters

Generic traits require special handling because Scala traits are compiled to JVM interfaces. A JVM class can contain fields, constructors, and ordinary methods, so a generic class can store reified type values directly in object fields. As discussed in Chapter 2, a JVM interface can contain abstract methods and, since Java 8, default methods, but it cannot contain per-instance fields. This difference matters for reification: a trait may declare or inherit methods whose types mention the trait type parameters, but the trait itself has no object layout in which those reified type values can be stored.

The problem is especially visible for default methods. A default method is implemented in the trait's interface classfile, not in each concrete subclass. This means that any metadata attached to the default method is in the interface classfile. If the default method returns a value of trait type parameter `A`, the type hint of which is, say `K(0)`, the method body is owned by the trait and its hint is interpreted in the trait layout; in this case, the reified VM has to resolve `K(0)` from the trait's point of view. However, the actual reified

type value for `A` is stored in, or computable from, the concrete class that implements the trait. The trait therefore needs a uniform way to retrieve that value from its receiver.

Consider the following trait.

Source code

```
trait Container[A] {
  def value: A
  def getValue: A = value
}
class IntContainer(v: Int) extends
↪  Container[Int] {
  def value: Int = v
}
val ic: IntContainer = new IntContainer(42)
val v: Int = ic.value
val gv: Int = ic.getValue
```

Erased code before backend

```
trait Container {
  def value: Object
  def getValue: Object = value
}
class IntContainer(v: Int) extends
↪  Container { // type parameter Int is
↪  erased
  def value: Int = v
}
val ic: IntContainer = new IntContainer(42)
val v: Int = ic.value
val gv: Int = Int.unbox(ic.getValue)
```

One possible design would be to avoid adding any reified member to the trait and rely only on fields in concrete subclasses. That alternative works when all useful code is implemented in concrete classes, but it fails for default methods. The default method `getValue` is declared in the trait classfile; the VM executing the code cannot directly refer to a subclass field. It needs an operation available through the trait interface itself.

The design, therefore, represents reified type values for trait type parameters using accessor methods instead of fields. For each type parameter that is available from a trait, the compiler generates a synthetic abstract accessor method in the trait. The accessor method has no parameters and returns a `ReifiedValue`.

Conceptually, the transformed trait has the following form. This accessor is abstract in the trait. The concrete subclass that implements the trait is responsible for providing an implementation of the accessor method.

Each trait-related type parameter is assigned a trait-level index, and the source-level meaning of erased values can be described by hints such as `K(i)`. This design follows the same principle as ordinary Scala trait implementations: the trait defines what information is required, and the concrete subclass provides the implementation. The trait only needs a way to access the reified type value.

```
trait Container {
  def reifiedTrait$Container$A: ReifiedValue
  def value: Object
  def getValue: Object = value
}
class IntContainer(v: Int) extends Container {
  def reifiedTrait$Container$A: ReifiedValue = IntTag
  def value: Int = v
}
val ic: IntContainer = new IntContainer(42)
val v: Int = ic.value
val gv: Int = Int.unbox(ic.getValue)
```

In this example, the desired reified unboxed semantics can be described by the following hints:

in trait `Container` hint `K(0)` corresponds to	:	abstract method `reifiedTrait$Container$A`
method `Container.value` return type	:	`K(0)`
default method `getValue` return type	:	`K(0)`
unboxing for `ic.getValue`	:	unnecessary

In executions, many times, the accessor may not be needed directly. If a generic trait only declares abstract methods and all useful implementation happens in the concrete subclass, then the concrete subclass can simply use its own reified fields. However, the presence of default methods make the accessor necessary, since default methods may need to refer to the reified type values for trait type parameters.

For example, the `IntContainer` class implements the abstract method `value` from trait `Container`, so the actual implementation of `value` is in the classfile of `IntContainer`. So a reified VM executing the method `value` is able to get the reified type value for `Container.A` from the field in `IntContainer`.

However, the default method `getValue` is defined in the trait classfile. After Java 8, the JVM supports default methods in interfaces, so Scala trait default methods are compiled directly into interface default methods.

When a reified VM executes the method `getValue` for class `IntContainer`, since the implementation of `getValue` is in the interface, it cannot immediately refer to the reified fields of `IntContainer`. Instead, it needs to know the reified type value of the trait type parameter `A`. The accessor method `reifiedTrait$Container$A` provides a bridge between the two locations. The reified VM knows to invoke the accessor method, and it also knows the receiver object, which is the concrete subclass, so it can retrieve the reified type value for `A` – a field of the concrete subclass – and use it to interpret the erased value returned by `getValue`.

3.8 Captured Type Parameter Layout

Previous sections described the simple cases: a method receives reified type values for its own type parameters, and a class stores reified type values for its own type parameters. Scala programs also contain nested definitions. A nested method, local class, or closure may refer to type parameters introduced by enclosing methods or classes. These type parameters, therefore, need to be represented as well. The nesting can be arbitrarily deep, with arbitrary combinations of classes, traits, and methods. Therefore, the compiler needs a way to assign indices to all type parameters that are visible in a given scope, including both owned and captured type parameters.

The central difficulty is that Scala nesting is not preserved directly in JVM bytecode. Local methods are compiled as separate methods of the enclosing class, and local classes are compiled as separate JVM classes.

Nested methods. Since in the JVM, methods are not nested, nested Scala methods are lifted into separate JVM methods on the enclosing class. Consider a local method whose parameter and return type both use the type parameter of the enclosing method.

Source code

```
def outer[T](x: T): T = {
  def inner(y: T): T = y
  inner(x)
}
```

Bytecode-level view

```
public Object outer(Object x){
  return inner$1(x);
}
private Object inner$1(Object y){
  return y;
}
```

At the source level, `inner` is nested inside `outer`. However, this nesting is not preserved directly in JVM bytecode. After lowering and erasure, the compiler emits two methods in the same JVM classfile. The local method is lifted to a separate bytecode-level method.

Reified code before backend

```
def outer(reified$outer$T: ReifiedValue)(x: Object): Object = {
  inner$1(reified$outer$T)(x)
}
def inner$1(reified$outer$T: ReifiedValue)(y: Object): Object = {
  // captures T from outer
  y
}
```

The important point is that `inner` still depends on `outer`'s type parameter `T`. In the reified translation, `outer` receives a reified type value for `T`. Since `inner` is compiled as a separate method, it cannot implicitly access that reified type value through source-level nesting. `inner` must explicitly capture the type parameter from `outer` and receive a reified type value for it as an additional parameter.

Nested classes. In the JVM, each classfile contains a single class, so local classes are compiled into separate classfiles. Thus, a similar issue appears for local classes when accessing reified type values of enclosing methods or classes. One difference is that a local class may outlive the activation of the enclosing method, so the reified type value cannot merely be passed to a method call. It must be stored in the generated class.

Source code

```
class Foo {
  def outer[T](x: T): AnyRef = {
    class Inner {
      def get: T = x
    }
    new Inner
  }
}
```

Bytecode-level view

```
// here, each class are in its own separate classfile
class Foo {
  public Object outer(Object x) {
    return new Foo$Inner(x);
  }
}
class Foo$Inner {
  private final Foo $outer;
  private final Object x;
  public Foo$Inner(Foo $outer, Object x) {
    this.$outer = $outer;
    this.x = x;
  }
  public Object get() {
    return this.x;
  }
}
```

The method `get` is declared inside the local class `Inner`, but its return type is the type parameter `T` declared by the enclosing method `outer`. The JVM does not represent `Inner` as a nested source-level class inside `outer`. Instead, it emits a separate classfile for the local class.

Although the erased descriptor of `get` returns `Object`, its source-level return type is still `Foo.outer.T`. The local class must therefore capture the type parameter from the enclosing method.

Reified code before backend

```
class Foo {
  def outer(reified$outer$T: ReifiedValue)(x: Object): AnyRef = {
    val reifiedLocal$outer$T: ReifiedValue = reified$outer$T
    new Foo$Inner(reifiedLocal$outer$T)(x)
  }
}
class Foo$Inner(reifiedField$outer$T: ReifiedValue)(x: Object) {
  // captures T from outer
  val reifiedField$outer$T: ReifiedValue = reifiedField$outer$T
  def get: Object = x
}
```

The design uses two capture rules.

Method capture rule. A method captures type parameters from enclosing methods up until the nearest enclosing class or trait. Class and trait type parameters are not copied into the method-level layout because they can be accessed through the receiver object's class or trait layout.

Class capture rule. A class captures type parameters from enclosing owners until the top-level owner, for example, a top-level object or package-level owner, is reached. This includes its own type parameters, type parameters from enclosing classes or traits, and type parameters from enclosing methods.

The following example illustrates the capture rules.

Source code

```
class C[A](a: A) {
  def outer[T](t: T): Any = {
    def inner[V](v: V): (A, T, V) = (a,
    ↪  t, v)
    class Local[B](b: B) {
      def useA: A = a
      def useT: T = t
      def useB: B = b
      def useInner[U](u: U): (A, T, U) =
      ↪  inner[U](u)
    }
    new Local[String]("hello")
  }
}
```

Bytecode-level view

```
// in one classfile
class C {
  Object a;
  public C(Object a) { this.a = a; }
  Object outer(Object t) {
    return new C$Local(this, t, "hello");
  }
  Object outer$inner$1(Object v, Object t) { ... }
}
// in another classfile
class C$Local {
  C outer$;
  Object b;
  public Local(C outer$, Object t, Object b) {
    this.outer$ = outer$;
    this.b = b;
    this.t = t;
  }
  Object useA() { return outer$.a; }
  Object useT() { return this.t; }
  Object useB() { return this.b; }
  Object useInner(Object u) {
    return outer$.outer$inner$1(u, this.t);
  }
}
```

This bytecode-level view shows why capture is necessary. The local method `inner` is no longer nested inside `outer`. If `inner` needs the type parameter `T` from `outer`, then `outer` must pass the reified type value for `T` as an extra argument. The local class `Local` is also a separate class, so it must store the reified type values for `B`, `T`, and `A` in its own object layout.

This example has five parameters:

`A`	:	introduced by class `C`
`T`	:	introduced by method `outer`
`V`	:	introduced by method `inner`
`U`	:	introduced by method `useInner`
`B`	:	introduced by local class `Local`

Reified code before backend

```
// in one classfile
class C(reifiedField$C$A: ReifiedValue)(a: Object) {
  val reifiedField$C$A: ReifiedValue = reifiedField$C$A
  def outer(reified$outer$T: ReifiedValue)(t: Object): Any = {
    val reifiedLocal$C$Local$B: ReifiedValue = ReferenceTag
    val reifiedLocal$outer$T: ReifiedValue = reified$outer$T
    val reifiedLocal$C$A: ReifiedValue = reifiedField$C$A
    new C$Local(reifiedLocal$C$Local$B, reifiedLocal$outer$T, reifiedLocal$C$A)("hello")
  }
  def outer$inner$1(reified$outer$T: ReifiedValue, reified$inner$V: ReifiedValue)(v: Object):
  ↪ Object
    = ... // captures T from outer
}
// in another classfile
class C$Local(reifiedField$Local$B: ReifiedValue, reifiedField$outer$T: ReifiedValue,
↪ reifiedField$C$A: ReifiedValue)(b: Object) {
  // captures A from C; T from outer
  val reifiedField$Local$B: ReifiedValue = reifiedField$Local$B
  val reifiedField$outer$T: ReifiedValue = reifiedField$outer$T
  val reifiedField$C$A: ReifiedValue = reifiedField$C$A
  def useA: Object = ...
  def useT: Object = ...
  def useB: Object = ...
  def useInner(reified$useInner$U: ReifiedValue)(u: Object): Object = ...
}
```

Class `C`. Class `C` introduces type parameter `A`, and it does not capture any type parameters from enclosing scopes, since it is a top-level class. Thus, type hints have the following layout[1]:

in class `C` hint `K(0)` corresponds to	:	field `reifiedField$C$A`
`A`	:	`K(0)`

Any method of `C` can refer to `A` through the receiver object. Therefore, `A` does not need to be copied to a method-level index slot `M(i)`.

Method `outer`. Method `outer` introduces type parameter `T`. According to the method capture rule, it captures no type parameters.

[1]The presentation of type hints in this subsection only focuses on representations for `M(n)` and `K(n)`. Other hints are present, for example, method parameters and return types, unnecessary boxings; they are not shown for simplicity.

Method `outer` is also a member of class `C`, so it may use the class type parameter `A`. However, `A` is not captured as a method type parameter, since it can be accessed through the receiver object. The type hints for method `outer` have the following layout:

in class `C` hint `K(0)` corresponds to	:	field `C.reifiedField$C$A`
in method `outer` hint `M(0)` corresponds to	:	parameter `reified$outer$T`
`A` from class `C`	:	`K(0)`
`T` from method `outer`	:	`M(0)`

Method `inner`. (`outer$inner$1`) Method `inner` is nested inside method `outer`. It introduces its own type parameter `V`. According to the method capture rule, `inner` captures method type parameters from enclosing methods up until the nearest enclosing class or trait. Therefore, `inner` captures `T` and receives reified type values for both `T` and `V` as method parameters.

The method `inner` may also refer to the class type parameter `A`, but `A` is not captured; it remains a class-level value available through the receiver object. The type hints for method `inner` have the following layout:

in class `C` hint `K(0)` corresponds to	:	field `C.reifiedField$C$A`
in method `inner` hint `M(0)` corresponds to	:	parameter `reified$inner$V`
in method `inner` hint `M(1)` corresponds to	:	parameter `reified$outer$T`
`A` from class `C`	:	`K(0)`
`T` from method `outer`	:	`M(1)`
`V` from method `inner`	:	`M(0)`

It is worth mentioning now that the same source-level type parameter may be represented differently depending on the context. For example, the type parameter `T` is represented as `M(0)` in the context of method `outer`, but it is represented as `M(1)` in the context of method `inner`.

Class `Local`. (`C$Local`) The class `Local` is nested inside method `outer`. It introduces its own type parameter `B`, but it also uses the type parameters `A` and `T` from class `C` and method `outer`. Unlike method `inner`, class `Local` creates object instances, so any type information needed by methods of `Local` must remain available for the lifetime of each `Local` instance.

According to the class capture rule, `Local` captures all type parameters from enclosing owners until the first static owner. So, it captures `T` from method `outer`, and `A` from class `C`. The type hints for class `Local` have the following layout:

in class `Local` hint `K(0)` corresponds to	:	field `Local.reifiedField$Local$B`
in class `Local` hint `K(1)` corresponds to	:	field `Local.reifiedField$outer$T`
in class `Local` hint `K(2)` corresponds to	:	field `Local.reifiedField$C$A`
`B` from class `Local`	:	`K(0)`
`T` from method `outer`	:	`K(1)`
`A` from class `C`	:	`K(2)`

Method `useA`, `useT`, `useB` Although these methods do not introduce new type parameters, they may still need to refer to some type parameter of their enclosing scope. According to the method capture rule, they capture method type parameters from enclosing methods up until the nearest enclosing class or trait. So they do not capture any type parameters. They can access the reified type values for `A`, `T`, and `B` through the receiver object.

Method `useInner`. Method `useInner` introduces its own type parameter `U`, but it does not capture any method type parameters.

in class `Local` hint `K(0)` corresponds to	:	field `Local.reifiedField$Local$B`
in class `Local` hint `K(1)` corresponds to	:	field `Local.reifiedField$outer$T`
in class `Local` hint `K(2)` corresponds to	:	field `Local.reifiedField$C$A`
in method `useInner` hint `M(0)` corresponds to	:	`reified$useInner$U`
`B` from class `Local`	:	`K(0)`
`T` from method `outer`	:	`K(1)`
`A` from class `C`	:	`K(2)`
`U` from method `useInner`	:	`M(0)`

Summary of the Example The full hint layout and the hints for the source-level meanings of values in the example are summarized in the figure 3.1.

The example shows the core design: `M(i)` is used for type parameters associated with method invocations; `K(i)` is used for type parameters associated with object instances. A nested method captures enclosing method type parameters as `M(i)`, but it does not cross the class boundary to capture class type parameters as class type parameters are available through the receiver object. A nested class, on the other hand, captures all reachable type parameters it needs into its object-level so the information is available after object construction.

The compiler internally maintains where each captured type parameter originates from, since it uses the information to generate the correct reified type values. A captured method type parameter must be passed from the enclosing method's argument list, while a captured class type parameter must be read from the outer object's reified field. However, the reified VM does not need to understand the ownership rules of type parameters, it only sees the compact hint representation `M(i)` and `K(i)`. This keeps the runtime interface simpler, smaller, and avoids requiring the runtime to understand Scala's lexical ownership rules.

Reified code before backend

```
1  // in one classfile
2  class C(reifiedField$C$A: ReifiedValue)(a: Object) {
3    val reifiedField$C$A: ReifiedValue =
     ↪  reifiedField$C$A
4    def outer(reified$outer$T: ReifiedValue)(t:
     ↪  Object): Any = {
5      val reifiedLocal$C$Local$B: ReifiedValue =
       ↪  ReferenceTag
6      val reifiedLocal$outer$T: ReifiedValue =
       ↪  reified$outer$T
7      val reifiedLocal$C$A: ReifiedValue =
       ↪  reifiedField$C$A
8      new C$Local(reifiedLocal$C$Local$B,
       ↪  reifiedLocal$outer$T,
       ↪  reifiedLocal$C$A)("hello")
9    }
10   def outer$inner$1(reified$outer$T: ReifiedValue,
     ↪  reified$inner$V: ReifiedValue)(v: Object):
     ↪  Object
11     = ... // captures T from outer
12 }
13 // in another classfile
14 class C$Local(reifiedField$Local$B: ReifiedValue,
   ↪  reifiedField$outer$T: ReifiedValue,
   ↪  reifiedField$C$A: ReifiedValue)(b: Object) {
15   // captures A from C; T from outer
16   val reifiedField$Local$B: ReifiedValue =
     ↪  reifiedField$Local$B
17   val reifiedField$outer$T: ReifiedValue =
     ↪  reifiedField$outer$T
18   val reifiedField$C$A: ReifiedValue =
     ↪  reifiedField$C$A
19   def useA: Object = ...
20   def useT: Object = ...
21   def useB: Object = ...
22   def useInner(reified$useInner$U: ReifiedValue)(u:
     ↪  Object): Object = ...
23 }
```

Reified layout and type hints

Class `C`

`C` hint `K(0)`	:	field `reifiedField$C$A`
`A`	:	`K(0)`
field `a`	:	`K(0)`

Method `outer`

`C` hint `K(0)`	:	field `C.reifiedField$C$A`
`outer` hint `M(0)`	:	parameter `reified$outer$T`
`A` from class `C`	:	`K(0)`
`T` from method `outer`	:	`M(0)`
parameter `t`	:	`M(0)`

Method `inner`

`C` hint `K(0)`	:	field `C.reifiedField$C$A`
`inner` hint `M(0)`	:	parameter `reified$inner$V`
`inner` hint `M(1)`	:	parameter `reified$outer$T`
`A` from class `C`	:	`K(0)`
`T` from method `outer`	:	`M(1)`
`V` from method `inner`	:	`M(0)`
parameter `v`	:	`M(0)`

Class `Local`

`Local` hint `K(0)`	:	field `Local.reifiedField$Local$B`
`Local` hint `K(1)`	:	field `Local.reifiedField$outer$T`
`Local` hint `K(2)`	:	field `Local.reifiedField$C$A`
`B` from class `Local`	:	`K(0)`
`T` from method `outer`	:	`K(1)`
`A` from class `C`	:	`K(2)`
field `b`	:	`K(0)`
`Local.useA` return	:	`K(2)`
`Local.useT` return	:	`K(1)`
`Local.useB` return	:	`K(0)`

Method `Local.useInner`

`Local` hint `K(0)`	:	field `Local.reifiedField$Local$B`
`Local` hint `K(1)`	:	field `Local.reifiedField$outer$T`
`Local` hint `K(2)`	:	field `Local.reifiedField$C$A`
`useInner` hint `M(0)`	:	`reified$useInner$U`
`B` from class `Local`	:	`K(0)`
`T` from method `outer`	:	`K(1)`
`A` from class `C`	:	`K(2)`
`U` from method `useInner`	:	`M(0)`
parameter `u`	:	`M(0)`

Figure 3.1: Source-level nesting and the corresponding reified layout.

3.9 Inheritance

The previous section described how type parameters from enclosing scopes are captured. Inheritance adds another dimension to the problem. A method may be declared in a superclass or a trait, but it may be executed on an instance of a subclass. Therefore, each class or trait in the hierarchy has its own view of the type parameters that it requires.

A main design rule is that a class-level hint `K(i)` is interpreted relative to the class or trait where it is declared. A `K(0)` in a superclass and a `K(0)` in a subclass are not necessarily the same slot. They belong to different reified layouts. The relationship between the layouts is established when the subclass extends the superclass and supplies reified type values for the superclass type parameters.

Consider the following example:

Source code

```
class Parent[A](a: A){
  def parentValue: A = a
}
class Child[B](b: B) extends Parent[B](b) {
  def childValue: B = b
}
```

Erased code before backend

```
class Parent(a: Object){
  def parentValue: Object = a
}
class Child(b: Object) extends Parent(b) {
  def childValue: Object = b
}
```

Since class `Child` extends `Parent` with type parameter `B`, it must provide a reified type value for `Parent`'s type parameter `A`. Thus, the subclass connects its own class-level layout with the superclass layout by passing the reified type value for `B` to the superclass constructor. Methods `parentValue` and `childValue` both have method return hints `K(0)`, but the two hints are interpreted relative to different classes. The hint `K(0)` in `Parent` refers to the reified type value for `A` in the `Parent` layout, while the hint `K(0)` in `Child` refers to the reified type value for `B` in the `Child` layout. Conceptually, the transformed classes have the following form.

```
class Parent(reifiedField$Parent$A: ReifiedValue)(a: Object) {
  val reifiedField$Parent$A: ReifiedValue = reifiedField$Parent$A
  def parentValue: Object = a
}
class Child(reifiedField$Child$B: ReifiedValue)(b: Object) extends
↪  Parent(reifiedField$Child$B)(b) {
  val reifiedField$Child$B: ReifiedValue = reifiedField$Child$B
  def childValue: Object = b
}
```

The subclass does not modify the superclass layout; it simply provides the reified type value for the superclass type parameter when it extends the superclass. Once the superclass constructor receives the reified type value, methods declared in the superclass can interpret their hints using the superclass layout. For example, if a call to `parentValue` is executed on an instance of `Child`, the execution of `parentValue` interprets the return hint `K(0)` using the superclass layout.

Concrete subclasses are handled in the same way. For example:

Source code

```
class IntChild(x: Int) extends Parent[Int](x)
```

Erased code before backend

```
class IntChild(x: Int) extends Parent(x)
```

Reified code before backend

```
class IntChild(x: Int) extends Parent(IntTag)(x)
```

The class `IntChild` provides the reified type value `IntTag` for `Parent`'s type parameter `A`.

The inherited method `Parent.parentValue` still has return hint `K(0)` relative to `Parent`. For an `IntChild` object, that superclass slot for `K(0)` contains the reified type value `IntTag`, so the reified VM can interpret the erased return value of `parentValue` as an `Int` value.

Overriding follows the same owner-relative rule:

Source code

```
class OverrideChild[A, B](a: A, b: B)
  ↪ extends Parent[B](b) {
  override def parentValue: B = b
}
```

Erased code before backend

```
class OverrideChild(a: Object, b: Object)
  ↪ extends Parent(b) {
  override def parentValue: Object = b
}
```

Reified code before backend

```
class OverrideChild(reifiedField$OverrideChild$A: ReifiedValue, reifiedField$OverrideChild$B:
  ↪ ReifiedValue)(a: Object, b: Object) extends Parent(reifiedField$OverrideChild$B)(b) {
  val reifiedField$OverrideChild$A: ReifiedValue = reifiedField$OverrideChild$A
  val reifiedField$OverrideChild$B: ReifiedValue = reifiedField$OverrideChild$B
  override def parentValue: Object = b
}
```

The inherited method `Parent.parentValue` has return hint `K(0)` in the `Parent` layout. The overriding method `OverrideChild.parentValue` in `OverrideChild` has return hint `K(1)` in the `OverrideChild` layout. The two hints are interpreted relative to different owners. No matter which version of the method is executed, the VM resolves the method owner, and uses the corresponding layout to interpret the return value. For example, the execution of `parentValue` on an `OverrideChild` object will use the overriding method, thus the hint `K(1)` in the `OverrideChild` layout is used to interpret the erased return value.

This owner-relative design prevents inheritance from forcing some global renumbering of type parameters. If every inherited type parameter had to be merged into one flattened

layout, then adding a superclass or trait could change the meaning of existing indices in the subclass, and the logic for assigning indices would also be more complex. With the owner-relative design, each class or trait can maintain its own layout.

Traits require the same principle, but they do not store reified fields directly. Instead, as mentioned in section 3.7, they define accessor methods that the concrete subclass must implement. For traits inheriting from other traits, the same principle applies. The sub-trait implements its parent accessor methods by either forwarding to its own accessor method for the corresponding type parameter or assigning a reified type value directly. For example:

Source code

```
trait Func[A, B]{
  def apply(x: A): B
}
trait IntInputFunc[B] extends Func[Int, B]
↪ {
  override def apply(x: Int): B
}
```

Erased code before backend

```
trait Func{
  def apply(x: Object): Object
}
trait IntInputFunc extends Func {
  override def apply(x: Int): Object
}
```

Reified code before backend

```
trait Func{
  def reifiedTrait$Func$A: ReifiedValue
  def reifiedTrait$Func$B: ReifiedValue
  def apply(x: Object): Object
}
trait IntInputFunc extends Func {
  def reifiedTrait$IntInputFunc$B: ReifiedValue
  override def reifiedTrait$Func$A: ReifiedValue = IntTag
  override def reifiedTrait$Func$B: ReifiedValue = reifiedTrait$IntInputFunc$B
  def apply(x: Int): Object
}
```

Trait `Func` introduces two type parameters, `A` and `B`, which the trait represents using two abstract accessor methods. The sub-trait `IntInputFunc` introduces its own type parameter `B`, and it also extends `Func` with type argument `Int` for `A`. Therefore, it provides an implementation for the accessor method for `A` that returns `IntTag`, and since it introduces its own type parameter `B`, it creates a new accessor method `reifiedTrait$IntInputFunc$B`. Now it has to implement the accessor method for `Func.B`: `reifiedTrait$Func$B`. `IntInputFunc` can simply forward the call to `reifiedTrait$IntInputFunc$B`.

Multiple Inheritance. Scala classes have only one direct superclass, but they may inherit from multiple traits. Therefore, the reified layout design must also handle cases where one class or trait simultaneously provides reified type values for several parent traits. This situation is similar to ordinary trait implementation: each trait has its own view of the reified layout.

Consider a class that implements two generic traits:

Source code

```
trait Left[A] {
  def left: A
}
trait Right[B] {
  def right: B
}
class Both[X, Y](x: X, y: Y) extends
↪  Left[X] with Right[Y] {
  def left: X = x
  def right: Y = y
}
```

Erased code before backend

```
trait Left {
  def left: Object
}
trait Right {
  def right: Object
}
class Both(x: Object, y: Object) extends
↪  Left with Right {
  def left: Object = x
  def right: Object = y
}
```

The extends clause establishes the mapping from the concrete class layout to each inherited trait layout. Since `Both` extends `Left[X]`, the trait type parameter `Left.A` is implemented using the class type parameter `Both.X`. Likewise, `Right.B` is implemented using `Both.Y`. The transformed class has the following form.

```
trait Left {
  def reifiedTrait$Left$A: ReifiedValue
  def left: Object
}
trait Right {
  def reifiedTrait$Right$B: ReifiedValue
  def right: Object
}
class Both(reifiedField$Both$X: ReifiedValue, reifiedField$Both$Y: ReifiedValue)(x: Object, y:
↪  Object) extends Left with Right {
  val reifiedField$Both$X: ReifiedValue = reifiedField$Both$X
  val reifiedField$Both$Y: ReifiedValue = reifiedField$Both$Y
  override def reifiedTrait$Left$A: ReifiedValue = reifiedField$Both$X
  override def reifiedTrait$Right$B: ReifiedValue = reifiedField$Both$Y
  def left: Object = x
  def right: Object = y
}
```

Again, this shows that the idea of each class or trait having its own layout applies to multiple inheritance as well. The class `Both` has its own layout with two class-level indices for its type parameters, and it implements the accessor methods for the inherited traits by forwarding to its own fields.

Default methods use the same interpretation. If trait `Left` has a default method that needs to refer to the reified type value for `A`, the return hint of that method is `K(0)` relative to `Left`; similarly if trait `Right` has a default method that needs to refer to the reified type value for `B`, the return hint of that method is `K(0)` relative to `Right`. Even when both default methods are executed on an instance of `Both`, their hints are interpreted through different trait accessors.

```
trait Left {
  def reifiedTrait$Left$A: ReifiedValue
  def left: Object
  def getLeft: Object = left
}
trait Right {
  def reifiedTrait$Right$B: ReifiedValue
  def right: Object
  def getRight: Object = right
}
class Both{
  // same as before
}
```

When the VM executes the default method `getLeft` on an instance of `Both`, it figures out that the default method is declared in trait `Left`, so it uses the accessor method for `Left.A` to retrieve the reified type value for `A` from the concrete subclass, and uses that reified type value to interpret the erased return value of `getLeft`. The same applies to the default method `getRight`.

Multiple inheritance can also create diamond inheritance patterns. For example:

```
trait Base[A] {
  def base: A
}
trait Left[A] extends Base[A]
trait Right[A] extends Base[A]
class Diamond[T](t: T) extends Left[T] with Right[T]
```

A diamond pattern does not create any special problem for the design. Scala's type system also prevents certain inconsistent forms of repeated inheritance. For example, a

class cannot inherit the same generic parent trait twice with different type arguments, such as inheriting `Left[Int]` and `Left[String]` at the same time. Such a hierarchy would require one trait to have two different reified layouts, which is not possible. Since this situation is rejected by the type system, the design does not need to handle it.

Reified Locals. As discussed in section 3.6, the reified type values for objects are explicitly written in locals. The locals are then passed to the constructor of the class. When inheritance is involved, the superclass type parameters' reified type values are also explicitly written in locals, but they are not passed to the allocating object's constructor.

Consider the following example. The reified local `reifiedLocal$Box$A` gets its value from `reifiedLocal$NamedBox$A`, but it is not passed to the constructor of `NamedBox` at allocation. The local is just there for the reified VM to read when it executes the `NEW` instruction for `NamedBox` to determine its internal object layout. This local is necessary because Java bytecode separates `NEW` and `<init>`; more details are discussed in section 5.5.2.

Source code

```
1  class Box[A](value: A)
2  class NamedBox[N, A](name: N,
3    value: A) extends Box[A](value)
4  val b = new NamedBox[String, Double]("pi",
   ↪  3.14)
```

Reified code before backend

```
1  class Box(reifiedField$Box$A: ReifiedValue)(value: Object){
2    val reifiedField$Box$A: ReifiedValue = reifiedField$Box$A
3  }
4  class NamedBox(reifiedField$NamedBox$N: ReifiedValue,
   ↪  reifiedField$NamedBox$A: ReifiedValue)(name: Object, value:
   ↪  Object) extends Box(reifiedField$NamedBox$A)(value){
5    val reifiedField$NamedBox$N: ReifiedValue =
     ↪  reifiedField$NamedBox$N
6    val reifiedField$NamedBox$A: ReifiedValue =
     ↪  reifiedField$NamedBox$A
7  }
8  val reifiedLocal$NamedBox$N: ReifiedValue = ReferenceTag
9  val reifiedLocal$NamedBox$A: ReifiedValue = DoubleTag
10 val reifiedLocal$Box$A: ReifiedValue = reifiedLocal$NamedBox$A //
   ↪  not passed to the constructor
11 val b = new NamedBox(reifiedLocal$NamedBox$N,
   ↪  reifiedLocal$NamedBox$A)("pi", 3.14)
```

3.10 Reified Type Values to Bytecode Type Hints

Runtime reified type values are only useful if the reified VM can understand how they relate to erased bytecode. Therefore, the compiler also emits bytecode attributes that describe how erased bytecode values correspond to source-level generic types. These attributes serve as type hints that the reified VM can read to recover information about erased values.

Reified type values and type hints serve different roles. Reified type values are runtime values, such as `IntTag`, `BooleanTag`, or `ReferenceTag`, carried by method parameters, object fields, local variables, or trait accessors. Type hints are metadata attached to bytecode constructs. They describe how a bytecode value whose JVM descriptor has been erased

should be interpreted, where the VM should look for the runtime reified type value needed for that interpretation, and which boxing and unboxing operation can be ignored under the reified unboxed semantics.

Both are needed in the reified VM. A hint such as `M(0)` says that the erased value depends on the first method type parameter, but it does not say what the actual type argument is; the reified VM still needs the actual runtime reified type value. Conversely, a reified type value such as `IntTag` says that the type argument is `Int`, but it does not say which bytecode parameter, field or return value should be interpreted using that tag. The hint provides that connection.

The design records hints at several kinds of places in the bytecode, as shown in Table 3.4.

Hint location	Purpose
class-level hints	describe the number and storage of class-related reified type values
trait-level hints	describe trait accessor methods for reified type values
field hints	describe fields whose erased type corresponds to a reified type hint
method-level hints	describe the number of method type parameters
parameter hints	map erased method parameters to `M(n)`, `K(n)`, primitive, or array hints
return hints	map erased method return values to type hints
invocation hints	describe the type of a specific call result
allocation hints	describe which local variables contain reified type values for a `new` instruction
boxing hints	mark boxing or unboxing operations that may be redundant for a reified-aware runtime

Table 3.4: Overview of bytecode type hint locations.

For the running example:

Source code

```
class Box[A](val value: A) {
  def get: A = value
}
def makeBox[T](x: T): Box[T] =
  new Box[T](x)
val b: Box[Int] = makeBox[Int](42)
val v: Int = b.get
```

Reified code before backend

```
class Box(reifiedField$Box$A: ReifiedValue, value: Object) {
  val reifiedField$Box$A: ReifiedValue = reifiedField$Box$A
  def get: Object = value
}
def makeBox(reified$makeBox$T: ReifiedValue)(x: Object): Box =
  val reifiedLocal$Box$A = reified$makeBox$T
  new Box(reifiedLocal$Box$A, x)
val b: Box = makeBox(IntTag)(Int.box(42))
val v: Int = Int.unbox(b.get)
```

Its generated bytecode looks like the following (in a simplified form):

```
class Box {
  byte reifiedField$Box$A;
  Object value;
  Box(byte reifiedField$Box$A, Object value) {
    this.reifiedField$Box$A = reifiedField$Box$A;
    this.value = value;
    Object.<init>(this);
  }
  Object value() { return this.value; } // getter for field value
  Object get() { return this.value(); }
}
class foo{
  static Box b;
  static int v;
  static void <clinit>() {
    MODULE$ = new foo();
    b = MODULE$.makeBox(73, // IntTag
                        BoxesRunTime.boxToInteger(42));
    v = BoxesRuntime.unboxToInt(b.get());
  }
  Box makeBox(byte reified$makeBox$T, Object x) {
    byte reifiedLocal$Box$A = reified$makeBox$T;
    return new Box(reifiedLocal$Box$A, x);
  }
}
```

The compiler needs to express the following relationships between erased bytecode semantics and the desired reified unboxed semantics:

- class `Box` has one class type parameter `A`
- in class `Box`, type parameter `A` has hint `K(0)`
- field `reifiedField$Box$A` of class `Box` is the reified type value for hint `K(0)` within the class `Box`
- type of field `value` of class `Box` corresponds to the class type parameter `A`
- methods `get` and `value` of class `Box` return a value whose type corresponds to the class type parameter `A`
- at the call site of method `makeBox`, the `Int` 42 is boxed to an `Integer` object. The boxing is needed for a standard JVM, but it is unnecessary for a reified VM since the reified type value for `T` already indicates that the type argument is `Int`,
- method `makeBox` has one method type parameter `T`

- parameter `reified$makeBox$T` of method `makeBox` is the reified type value for the method type parameter `T`
- in method `makeBox`, type parameter `T` has hint `M(0)`
- the type of parameter `x` of method `makeBox` corresponds to the method type parameter `T`
- the allocation site `new Box[T]` uses the reified type value for `T` as the reified type value for class type parameter `A`, it is stored in JVM local `reifiedLocal$Box$A`
- at the call site of `b.get()`, the returned value is a primitive `int`
- since the return value of `b.get()` is primitive, the unboxing operation is no longer needed for the reified unboxed semantics

The detailed JVM bytecode encoding of these relationships is not part of this chapter. Chapter 5 describes the custom bytecode attributes used to attach this information to classes, fields, methods, and bytecode instructions.

Chapter 4

Compiler Transformation

Chapter 3 described the design of the reified type hint system. The design introduces reified type values and bytecode-level type hints so that a reified VM can recover selected generic type information after erasure. This chapter describes how the design is implemented in the Scala 3 compiler.

The implementation is organized as a set of compiler transformations. These transformations operate on the Scala compiler's internal representation of a program before it is lowered to JVM bytecode. At a high level, the compiler first inserts explicit reified type values into the typed Scala program. It then records source-level type information before erasure removes it. Later phases preserve this information through erasure and other lowering transformations until the backend emits the classfile metadata described in Chapter 5.

This chapter focuses on the compiler transformations. Specifically, it explains how the compiler modifies the internal intermediate representation (IR) and method or class information so that the backend has enough information to emit the classfile metadata.

4.1 Compiler Internal Representation

The Scala 3 compiler represents a program using several related internal structures [10]. Trees represent the syntax of the program, symbols identify program definitions, and symbol infos describe their types and signatures. Attachments allow compiler phases to associate additional metadata with trees. Each compiler phase rewrites these trees, updates associated symbol infos and types, and passes the transformed program to the next phase. This section briefly introduces these concepts, which are used by the compiler transformation presented in this thesis.

Trees. Trees form the compiler's IR of a Scala program. Each tree node represents a language construct, such as a class definition, method definition, method application,

field selection, or literal. For example, a method is represented by a `DefDef` tree, a value definition by a `ValDef`, and a method call by an `Apply` tree.

The parser initially produces untyped trees. During type checking, names are resolved and type information is assigned to the trees, producing typed trees. Subsequent compiler phases operate primarily on typed trees and gradually replace high-level Scala constructs with lower-level representations [10].

Symbols. A symbol uniquely identifies a definition within the compiler, such as a class, trait, method, field, local variable, or type parameter. Symbols provide a common identity for all references to the same definition. This is different from trees, which represent individual occurrences of program constructs. For example, a method definition has a symbol that is referenced by all calls to that method.

A symbol itself mainly provides identity. Its semantic properties, including its name, owner, compiler flags, and type information, are exposed through its denotation. Denotations are phase-dependent, allowing the compiler's view of a definition to change as the definition is transformed by different phases. Compiler transformations that introduce new definitions must therefore create corresponding symbols and assign them appropriate owners, flags, and type information.

Info. The info of a symbol describes the type or signature of the definition represented by that symbol. Infos are represented using the compiler's internal `Type` hierarchy. For example, the info of a value symbol is its value type. The info of a method may be represented by a `MethodType`, or a `PolyType` for polymorphic methods. The info of a class is represented by a `ClassInfo`, which records information such as the class's parents and declarations. Symbol infos may change between compiler phases. For example, a transformation that changes the parameters or result type of a method must update both the method tree and the method symbol's info. Otherwise, later phases may observe a tree whose structure is inconsistent with its symbol.

Attachments. Attachments are auxiliary values attached to compiler trees. They provide a typed key-value mechanism for recording information that does not naturally belong in the tree structure or in the tree's type. An attachment can be added, retrieved, or removed using its property key. Scala 3 also provides 'sticky' attachment keys, whose values are copied when their containing tree is copied.

Attachments are useful for communicating information between compiler phases without introducing additional language-level tree nodes or modifying symbol signatures. In this work, attachments are used to associate reification-related metadata with particular definitions or expressions so that the information is retrieved by later transformations or by the bytecode backend.

4.2 Scala Compiler Phases

The Scala 3 compiler is organized as a sequence of `Phases`. Each phase performs a particular transformation on the program's trees and, when necessary, updates the associated symbols and infos. A phase declares ordering constraints that determine which phases must execute before or after it. Early phases parse and type-check the source program. Later transformation phases lower high-level Scala constructs into simpler forms. The Erasure phase is in the middle and it performs type erasure on Scala types. Finally, the backend translates the resulting trees into JVM classfiles.

Many transformations are implemented as `MiniPhases`. Multiple mini-phases can be grouped and executed during a single traversal of the syntax tree, reducing compilation overhead while keeping individual transformations modular.

The reification implementation introduces and modifies several phases. The phases most relevant to reification are in Figure 4.1.

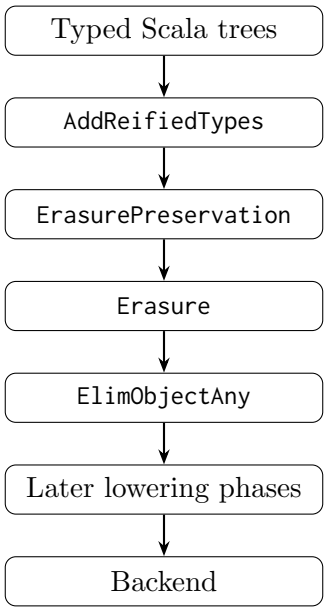

Figure 4.1: Placement of the reification phases in the Scala compiler pipeline involved in the reification transformation.

The central phase is `AddReifiedTypes`. It runs before erasure, while the compiler still has access to source-level generic type information. This phase inserts the reified type values into method calls, class constructors, fields, and traits.

The `ErasurePreservation` phase also runs before erasure. It records type hints that describe the source-level generic types of parameters, fields, return values, and invocation results. These hints are later used by the bytecode backend.

The ordinary Scala `Erasure` phase is then modified so that erased generic values can still be distinguished from ordinary `Object` values during the remaining `Erasure` phase. This distinction is represented using the internal type `ObjectAny`. A later phase, `ElimObjectAny`, removes this internal type and replaces it with ordinary `Object` before later lowering phases.

The final phase is the backend phase, which normally emits bytecode for Scala programs. In this work, the backend is modified to also emit the reified type hints as described in Chapter 5.

4.2.1 AddReifiedTypes

The `AddReifiedTypes` phase is responsible for inserting explicit reified type values into the program. It runs right before erasure because this is the latest point where the compiler still knows the original type arguments of generic classes and methods.

In particular, it updates three connected parts of the compiler IR:

- *symbol infos*, which describe the signatures of methods, constructors, fields, and generated members;
- *symbols*, including newly generated parameters, fields, local variables, and trait accessor methods;
- *trees*, which are rewritten to define and use the newly generated symbols.

Creating Symbols. This phase introduces generated symbols for reified type values. These symbols represent compiler created parameters, fields, local values, or accessors, which do not come directly from source code, but later phases treat them like ordinary program definitions.

Generated entity	Purpose
`reified$...`	Method parameter storing a reified type value for a method type parameter
`reifiedField$...`	Field storing a reified type value for a class type parameter
`reifiedLocal$...`	Local value holding a reified type value before a constructor call
`reifiedTrait$...`	Trait accessor method returning a reified type value for a trait type parameter

Table 4.1: Generated reified symbols.

These generated symbols are ordinary compiler symbols. They have owners, infos, flags, and corresponding trees. Later phases can therefore process them using the existing

compiler infrastructure. For example, a generated reified field is treated like a field by later phases, and a generated reified method parameter is treated like a parameter.

Transforming Trees. After the relevant symbols have been created, this phase rewrites trees to make reified type values explicit.

At the tree level, this includes inserting generated `ValDef` nodes, adding arguments to `Apply` trees, adding fields or accessors to class and trait templates, and replacing some type-dependent operations with operations that use reified type values. The result is still an ordinary typed Scala compiler tree. For example, a tree representing a generic call is changed so that the call includes reified arguments. A tree representing object allocation is changed so that the constructor receives the reified type values required by the allocated class.

The phase therefore turns implicit type information into explicit values. Before the phase, type arguments exist only in the typed representation of the program. After the phase, selected type arguments also have corresponding runtime values.

Transforming Infos. Adding a reified type value is not only a tree rewrite. The symbol's info must also be changed. Otherwise, the compiler would see a mismatch between the symbol declaration and its use cases.

For example, if a generic method is transformed so that it receives an additional reified type value, the compiler's method symbol info must describe the additional parameter list. Similarly, if a constructor is transformed to receive reified type values for class type parameters, the constructor info must be updated before constructor calls are rewritten. Adding fields or accessors for reified type values also requires updating the class info to include the new members.

Creating Reified Type Values. This phase needs a way to translate a Scala type into a reified type value. This translation is used at method call sites, constructor calls, trait accessor implementations, and array construction sites.

For primitive types, the translation is direct. For example, `Int` is translated to `IntTag`, and `Boolean` is translated to `BooleanTag`. For ordinary reference types, the translation produces `ReferenceTag`. A method type parameter is represented by a read of the reified method parameter. A class type parameter is represented by a read of the reified field or, for traits, by a call to the generated accessor method.

4.2.2 ErasurePreservation

The `AddReifiedTypes` phase makes type information available as runtime values. The backend must also know how erased bytecode positions correspond to source-level generic types. For this reason, the implementation includes a second phase, `ErasurePreservation`.

The purpose of `ErasurePreservation` is to record type-hint metadata immediately before the compiler erases generic types. At this point, the compiler still knows whether a type is a method type parameter, a class type parameter, a primitive type, an array type, or an ordinary reference type.

`ErasurePreservation` converts selected Scala types into the type-hint representation introduced in Section 3.4. For example, a method type parameter is converted into an `M(i)` hint, where `i` identifies the method type-parameter position. A class type parameter is converted into a `K(i)` hint, where `i` identifies the class type-parameter position in the relevant reified layout.

Method Type Parameter, Parameter, and Return Hints. `ErasurePreservation` records the generic meaning of method parameters and return values. This data is attached as Scala attachments to the relevant `DefDef` trees. Section 5.4 describes how this information is later emitted as classfile metadata.

Field Hints. `ErasurePreservation` also records the generic meaning of class fields. This data is attached as Scala attachments to the relevant `ValDef` trees. Section 5.3 describes how this information is later emitted as classfile metadata.

Invocation Result Hints. Method calls are another place where erased bytecode loses useful generic information. A call may return `Object` at the JVM level even though the source-level result type is a method or class type parameter. `ErasurePreservation` therefore records invocation result hints on call trees as Scala attachments. Section 5.5.1 describes the hint in more detail.

4.2.3 Erasure

Scala's erasure phase maps high-level Scala types to JVM-level types. It performs transformations on symbol infos.

For the reification transformation, a problem is that many different source-level types erase to `Object`. For example, explicitly written `Object` type, `Any` type, `AnyRef` type, and erased type parameter all have the same JVM representation. However, they are not treated the same by the reification system.

To preserve this distinction, the implementation introduces an internal marker type called `ObjectAny`. This type is used inside the compiler to represent an object-like value that came from generic erasure. It is not a source-level type and is not emitted as a separate JVM type. It is a temporary compiler marker. Now, instead of erasing all generic types to `Object`, the compiler erases them to `ObjectAny`. This distinction allows later compiler logic to recognize values that were produced by erasing a generic type parameter. After

the relevant transformations have used this information, the `ElimObjectAny` phase replaces `ObjectAny` with ordinary `Object`, restoring a standard JVM-level representation.

Boxing and Unboxing. Generic code involving primitive types often requires boxing and unboxing after erasure. In the running example from Section 3.3, the generic method `makeBox` is instantiated with `Int`. Thus, the call to `makeBox[Int]` requires boxing the `Int` value into an `Integer` object. Similarly, the call to `get` requires unboxing the `Integer` back into an `Int` when storing the result back to a Scala `Int` variable. The Erasure phase, therefore, introduces boxing and unboxing operations into the program tree.

These boxing and unboxing operations are required for ordinary JVM execution, but they are not required for the reified unboxed semantics. The modified compiler therefore introduces Scala attachments to the unboxing or boxing trees. These attachments serve as markers for the selected boxing and unboxing operations. They indicate that the operation was introduced as part of erased generic handling and should not be executed in the reified unboxed semantics. Later, the backend can emit classfile metadata that identify these operations to the VM.

Casts. Casts involving erased generic values also require special handling. A source-level upcast from a generic type parameter to `Object` must be preserved explicitly for the reified VM. For example, consider casting a value of type parameter `T` to target type `Object`. During erasure, `T` is erased to `Object`, so the source and target types are the same. The ordinary compiler can therefore remove the cast without emitting a corresponding JVM bytecode instruction. Removing the cast is valid under the standard erased JVM semantics, but it loses information required by the reified unboxed semantics. A value whose source-level type is `T` may be represented as an unboxed primitive, whereas the target type `Object` must use an object-reference representation. The reified VM must therefore know that the source program crossed from the representation of `T` to the representation of `Object`. This differs from an expression whose type was already `Object`, for which no such representation change is required.

The compiler preserves the otherwise erased cast as a reified cast through a reified runtime helper `reifiedAsInstanceOf`, allowing the reified VM to perform the appropriate representation conversion. The helper is a simple identity function that is implemented in the reified runtime library, and the reified VM intrinsifies it.

4.2.4 Preserving Reification In Other Transformations

Other than the phases `AddReifiedTypes`, `ErasurePreservation`, and `Erasure`, the program still passes through the normal phases in the Scala compiler. These phases may introduce new methods, move local definitions, copy trait methods, generate fields, or rewrite closures. Such transformations can easily lose or invalidate reification metadata if they are

not adjusted. The phases may also change the IR in ways that make it difficult for the backend to emit the correct classfile metadata. The implementation therefore modifies these phases so that they either preserve the reification information, or update the Scala IR so that the backend can still emit the correct metadata.

Generic Arrays. Generic array construction is another operation where erased JVM types are insufficient. In ordinary Scala, creating an array whose element type is a type parameter usually requires a `ClassTag`. In the reified compiler, the required type information may instead be obtained from the inserted reified type value.

Standard Scala uses a `ClassTag[T]` to retain enough runtime information to allocate `Array[T]`. In the modified compiler, this role is performed by the reified type value already associated with `T`. Array construction is therefore lowered to a runtime helper that uses the reified value to select the correct JVM array representation.

Expand Closures. Calls to Scala closures are transformed into a `invokeDynamic` bytecode instruction. Unlike other invoke instructions, `invokeDynamic` does not directly specify the target method. Instead, it represents a dynamic call site whose target is determined at runtime by a bootstrap method.

The classfile includes the necessary information for this process. This includes the bootstrap method, typically `LambdaMetafactory.metafactory`, and its arguments. When the JVM first executes the `invokeDynamic` instruction, it calls the bootstrap method to obtain an object. The object defines the actual method to be invoked. The bootstrap method is called only once, and later calls use the same object and the resolved target method directly.

This is a problem for the reified VM. Since this class is not available during compilation, the compiler cannot modify it for reification. To address this, the added `ExpandClosures` phase converts closures into explicit anonymous classes during compilation. Doing so allows the compiler to treat the closure as an ordinary nested class, and it can insert reified type values into the closure's constructor and methods. The call site of such a closure can therefore use other invoke instructions.

Lambda Lifting and Captured Values. The `LambdaLift` phase turns nested functions or local definitions into methods with additional parameters for captured variables.

The compiler must therefore update both the generated parameter lists and the associated type hint metadata. When a captured value becomes an explicit parameter, the method's parameter hints must be extended so that the backend can still interpret the lifted method correctly.

Mixin. The `Mixin` phase lowers trait members into concrete class members, creates accessors, and creates forwarding methods in the class. A forwarder is a method generated in the implementing class that delegates to a method defined by a trait.

When a trait method is forwarded, the corresponding type hints must also be remapped. A hint that refers to a trait type parameter from the trait's perspective needs to be translated into the implementing class's reified layout.

4.2.5 Backend Responsibilities

By the time the backend runs, the source program has been lowered toward JVM-shaped code. Although generic types have mostly been erased, the modified compiler has preserved the information needed by a reified-aware backend.

The backend can then serialize this information into the classfiles as metadata described in Chapter 5. Thus, Chapter 5 begins where this chapter ends: the compiler tree has already been rewritten, generic type information has been preserved as hints, and the backend's task is to encode those hints into bytecode.

Chapter 5

Encoding Reified Type Information in JVM Bytecode

The previous chapters described the design of a reified unboxed language. The reified unboxed language can be described in terms of a modified Scala compiler generated bytecode along with additional metadata on the bytecode. This chapter describes the bytecode-level encoding of reified type information. Previously, chapter 3 described how the modified Scala compiler introduces reified type values and propagates them through the compilation pipeline. Those explicit values make selected generic type information available at runtime. However, the values alone do not explain their role to a JVM. A reified VM must also know additional information, such as which values correspond to class type parameters, which values correspond to method type parameters, which bytecode instructions produce erased generic results, and which object allocations are associated with particular reified type arguments.

This thesis uses JVM classfile attributes as the bytecode-level interface between the modified Scala compiler and the reified VM. JVM classfiles support attributes as a general mechanism for storing metadata. Attributes can be attached to different parts of a classfile, including the class itself, fields, methods, and method bodies. Standard attributes are already used for many kinds of metadata. For example, `LineNumberTable` records the mapping between bytecode offsets and source line numbers, `LocalVariableTable` records local variable names and live ranges, `Signature` preserves generic signatures that are erased from JVM descriptors, and `RuntimeVisibleAnnotations` stores annotation metadata.

The same general mechanism can also be used for custom attributes. A custom attribute is not assigned a predefined meaning by the JVM specification, but it follows the same structural format as other classfile attributes. Standard JVMs can ignore attributes they do not understand, while a modified VM can recognize and interpret them. This makes custom classfile attributes a suitable interface between the modified Scala compiler and the reified VM.

The information required by the runtime is distributed across multiple levels of the

generated classfile. Some information belongs to a class, such as the number of type parameters of a generic class, the fields that store their reified type values, and the accessor method used to retrieve them. Some information belongs to a field, such as the source level type of an erased field. Some information belongs to a method, such as the type hints for method parameters and return values. Other information belongs to specific bytecode instructions, such as the return type of an invocation or the reified arguments associated with a `NEW` instruction.

Therefore, the modified compiler emits a family of custom attributes rather than a single global attribute. The explicit reified type values carry runtime data, while the custom attributes describe the meaning and location of that data to the runtime. The compiler computes the relevant type hints before erasure removes source-level generic information, and the JVM backend then emits these hints into the generated classfiles.

One benefit of this design is that it separates program execution from attribute interpretation. The transformed bytecode program still contains ordinary bytecode instructions and explicit reified type values. The custom attributes do not replace these values; instead, they describe their meaning. A standard JVM can execute the bytecode without interpreting the custom attributes, while a reified VM can use the attributes to perform reification or specialization.

5.1 Overview

The compiler emits custom attributes at four levels of the JVM classfile: class level, field level, method level, and code level, as shown in Table 5.1. Each level corresponds to a different part of the classfile structure and describes different aspects of the reified type information.

Class-level attributes describe type information associated with a class or trait. For ordinary generic classes, this includes the number of class type parameters and the fields that store their reified type values. For generic traits, it includes accessor methods used to retrieve reified type values from implementing classes.

Field-level attributes describe type information associated with individual fields. This is necessary because fields whose source-level types mention type parameters are erased in the JVM descriptor. A field whose source type is `T` is emitted with descriptor `java.lang.Object` after erasure, but the runtime still needs to know that the field corresponds to a class type parameter. The `FieldType` attribute records this pre-erasure type hint directly on the generated field.

Method-level attributes describe type information associated with a method declaration. This includes the number of method type parameters, the type hints for method parameters, and the type hint for the return value.

Level	Attribute	Purpose
Class	`ClassTypeParameterCount`	Number of type parameters represented by the class or trait
Class	`ClassTypeParamList`	Reified fields corresponding to class type parameters
Class	`TraitTypeParamList`	Reified accessor methods corresponding to trait type parameters
Field	`FieldType`	Type hint for an erased field
Method	`MethodTypeParameterCount`	Number of method type parameters
Method	`MethodParameterType`	Type hints for method parameters
Method	`MethodReturnType`	Type hint for the method return value
Code	`InvokeReturnType`	Type hints for individual invocation results
Code	`BCNewTypeArgs`	Reified type arguments associated with `NEW` instructions
Code	`ExtraBoxUnbox`	Boxing and unboxing instructions that can be treated specially by the runtime

Table 5.1: Custom attributes emitted by the compiler.

Code-level attributes describe type information associated with specific bytecode instructions. For example, an invocation may return `java.lang.Object` after erasure even though its source-level return type was a type parameter. Similarly, a `NEW` instruction may allocate an object whose generic type arguments are represented by reified type values stored in local variables; a code attribute can record the mapping between the `NEW` instruction and the relevant reified type values.

For the rest of this chapter, custom attributes are written with an at-sign prefix, such as `@FieldType` or `@MethodReturnType`.

All custom attributes follow the general JVM attribute structure:

```
attribute{
  u2 attribute_name_index;
  u4 attribute_length;
  u1 info[attribute_length]; // the content of the attribute
}
```

The `attribute_name_index` points into the constant pool and names the attribute. The `attribute_length` gives the size of the attribute content. The remaining bytes are the

content of the attribute, which can be structured in any way that the compiler and runtime agree on. The structure of the content is not defined by the JVM specification; it is determined by the needs of the reification transformation.

The attributes in this chapter often contain type hints. The type hints are the `TypeB` enumeration introduced in Section 3.4.

```
TypeHint = Primitive(p)
         | Reference
         | M(i)
         | K(i)
         | Array(TypeHint)
         | None
         | Receiver
```

5.2 Class-level Attributes

Class-level attributes describe reified type information associated with a class or trait. This information belongs to the object's runtime representation and may be used by multiple instance methods after construction. There are three class-level attributes:

- `ClassTypeParameterCount`, which records the number of class type parameters represented by a class or trait;
- `ClassTypeParamList`, which records the fields that store reified type values for a generic class;
- `TraitTypeParamList`, which records the accessor methods that return reified type values for a generic trait.

Consider the bytecode for the reified Scala program for the class `Pair` seen in Section 3.6.

```
class Pair extends java.lang.Object{
  private final byte reified$Pair$A;
    descriptor: B;
  private final byte reified$Pair$B;
    descriptor: B;
  private final java.lang.Object fst;
    descriptor: Ljava/lang/Object;
  private final java.lang.Object snd;
    descriptor: Ljava/lang/Object;

  public Pair(byte, byte, java.lang.Object, java.lang.Object);
    descriptor: (BBLjava/lang/Object;Ljava/lang/Object;)V
  public java.lang.Object fst();
    descriptor: ()Ljava/lang/Object;
  public java.lang.Object snd();
    descriptor: ()Ljava/lang/Object;
}
```

Figure 5.1: Generated JVM bytecode for the reified Scala class `Pair[A, B]`.

Fields `fst` and `snd` have types `java.lang.Object` in their JVM descriptor because type parameters `A` and `B` have been erased. The fields `reified$Pair$A` and `reified$Pair$B` are not part of the original source program; they are inserted by the reification transformation and store the reified type values of the class type parameters `A` and `B`. The class-level attributes describe these inserted fields to the runtime by recording how many represented type parameters exist and where their reified type values are stored.

Trait type parameters also need class-level attributes, although they use a slightly different representation. For traits, the reified type value is retrieved through an accessor method rather than an instance field.

5.2.1 ClassTypeParameterCount

The `ClassTypeParameterCount` attribute records the number of type parameters represented at the class level. This includes class type parameters of a generic class, as well as captured type parameters represented at the class level. The attribute has the following structure:

```
ClassTypeParameterCount_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 count;
}
```

For a non-generic class that does not capture any type parameters, the attribute is omitted. For a class with type parameters, the attribute is present. For example, for the class `Pair[A, B]`, the generated classfile contains:

```
class Pair extends java.lang.Object{
    @ClassTypeParameterCount {
      count = 2
    }
}
```

The count describes the size of the class-level `K` index space. The count also includes any captured type parameters represented at the class level, for example when a nested class captures type information from an outer context.

5.2.2 ClassTypeParamList

The `ClassTypeParamList` attribute records the fields that store reified type values for a generic class. Each entry corresponds to one represented class type parameter.

```
ClassTypeParamList {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 class_type_param_num;
  u2 type_param_field_ref_index[class_type_param_num];
}
```

Each `type_param_field_ref_index` entry is a constant-pool reference to a field that stores the reified type value for a class type parameter. The i-th entry gives the field that stores the reified type value for `K(i)`. For the class `Pair[A, B]`, a generated shape is:

```
class Pair extends java.lang.Object{
  ...
    @ClassTypeParameterCount {
      count = 2
    }
    @ClassTypeParamList {
      class_type_param_num = 2
      0 -> Fieldref Pair.reifiedField$Pair$A:B
      1 -> Fieldref Pair.reifiedField$Pair$B:B
    }
}
```

This mapping connects each class level type parameter to its corresponding generated field. A runtime can therefore determine that `K(0)`, corresponding to `A`, is represented by the reified field `reifiedField$Pair$A`; and `K(1)`, corresponding to `B`, is represented by the reified field `reifiedField$Pair$B`.

5.2.3 TraitTypeParamList

The `TraitTypeParamList` attribute is the trait analogue of `ClassTypeParamList`. It records accessor methods rather than fields.

```
TraitTypeParamList {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 class_type_param_num;
  u2 type_param_accessor_ref_index[class_type_param_num];
}
```

Each `type_param_accessor_ref_index` entry is a constant-pool reference to an interface method that returns the reified type value for a trait type parameter. The i-th entry gives the method that returns the reified type value for `K(i)`.

Consider the `Container[A]` trait from Section 3.7.

```
1  interface Container{
2      public abstract byte reified$Container$A();
3          descriptor: ()B
4      public abstract java.lang.Object value();
5          descriptor: ()Ljava/lang/Object;
6      public default java.lang.Object getValue();
7          descriptor: ()Ljava/lang/Object;
8      @ClassTypeParameterCount {
9        count = 1
10     }
11     @TraitTypeParamList {
12       trait_type_param_num = 1
13       0 -> Methodref Container.reified$Container$A:()B
14     }
15 }
```

Figure 5.2: JVM bytecode for the reified Scala trait `Container[A]`.

The `@TraitTypeParamList` attribute maps each trait type-parameter index to the accessor method that returns its reified type value. The reified VM at runtime finds the abstract accessor method and dispatches to the implementation provided by the concrete class, thus obtaining the reified type value for the trait type parameter.

5.3 Field-level Attributes

Field-level attributes describe reified type information for a field in a class. From the class-level attributes, the runtime knows how many class type parameters are represented and which fields or accessor methods store their reified type values. However, it still needs to know the source-level meaning of ordinary erased fields. For example, for the `Pair[A, B](fst: A, snd: B)` class, its `fst` and `snd` fields are compiled with erased field descriptors:

```
1  private final java.lang.Object fst;
2    descriptor: Ljava/lang/Object;
3  private final java.lang.Object snd;
4    descriptor: Ljava/lang/Object;
```

The descriptor only says that the field contains an object reference. The class-level attribute `ClassTypeParameterCount` says that there are two class type parameters, and the class-level attribute `ClassTypeParamList` says that the reified type values for the type parameters are stored in particular fields. They do not say that the field corresponds to the class type parameter `A` or `B`. The field-level attribute `FieldType` provides this missing information by recording a type hint directly on the field.

5.3.1 FieldType

The `FieldType` attribute records the type hint for an erased field.

```
FieldType {
  u2 attribute_name_index;
  u4 attribute_length;
  TypeHint field_type;
}
```

In practice, fields are associated with class or trait type parameters rather than method type parameters, so this attribute typically uses the `K(i)` form.

For the field `t` in `Box[T]`, the compiler emits a field descriptor of `Ljava/lang/Object;` after erasure, but it also emits `@FieldType(K0)` to record that the source-level field corresponds to the first class type parameter.

```
1  private final java.lang.Object t;
2    descriptor: Ljava/lang/Object;
3    @FieldType(K(0))
```

For the earlier example of `Pair[A, B](val first: A, val second: B)`, the compiler emits:

```
1  private final java.lang.Object first;
2    descriptor: Ljava/lang/Object;
3    @FieldType(K(0))
4  private final java.lang.Object second;
5    descriptor: Ljava/lang/Object;
6    @FieldType(K(1))
```

5.4 Method-level Attributes

Method-level attributes describe generic relationships at method declarations. They are attached to the method itself, not to a particular call site. They describe how the erased method descriptor relates to the source-level method signature. There are three method-level attributes:

- `MethodTypeParameterCount`, which records the number of method-level type parameters;
- `MethodParameterType`, which records the type hints for method parameters;
- `MethodReturnType`, which records the type hint for the method return value.

Consider the `choose[A, B](x: A, y: B): A` method from Section 3.5.

```
1  public java.lang.Object choose(byte, byte, java.lang.Object, java.lang.Object);
2    descriptor: (BBLjava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
```

Figure 5.3: JVM bytecode for the reified Scala method `choose[A, B](x: A, y: B): A`.

The two leading `byte` parameters are the compiler inserted reified type values for `A` and `B`. The ordinary parameters and return type are erased to `java.lang.Object`. The method-level attributes describe the generic relationships between the erased method descriptor and the source-level method signature.

5.4.1 MethodTypeParameterCount

The `MethodTypeParameterCount` attribute records the number of method-level type parameters represented by a method.

```
MethodTypeParameterCount {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 count;
}
```

For a method that has no type parameters and does not capture any, the attribute is omitted. For a method with type parameters, e.g, the `choose` method above, the generated classfile contains:

```
1  public java.lang.Object choose(byte, byte, java.lang.Object, java.lang.Object);
2    descriptor: (BBLjava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
3    @MethodTypeParameterCount {
4      count = 2
5    }
```

The count describes the size of the method-level `M` index space. The count may also include captured type parameters represented at the method level. The VM can use the count and some agreed convention for how the compiler orders method reified type values and usual method parameters to fetch the reified type values for method type parameters at runtime. In this implementation, the method reified type values are at the front of the parameter list.

5.4.2 MethodParameterType

The `MethodParameterType` attribute records the type hints for method parameters.

```
MethodParameterType {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 parameter_num;
  TypeHint parameter_type[parameter_num];
}
```

The entries describe ordinary value parameters. For the compiler inserted reified type values, it simply describes them as type hint `Byte`. For the `choose` method, the generated classfile contains:

```
public java.lang.Object choose(byte, byte, java.lang.Object, java.lang.Object);
  descriptor: (BBLjava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
  @MethodParameterType {
    parameter_num = 4
    0 -> Byte
    1 -> Byte
    2 -> M(0)
    3 -> M(1)
  }
```

Here, the hints show that the third parameter has a type hint `M(0)`, and the fourth parameter has the type hint `M(1)`, which means that the source-level types of the two parameters are the first and second method type parameters, respectively. Together with the `MethodTypeParameterCount` attribute, the runtime can determine that the first two parameters are the compiler inserted reified type values for the method type parameters, and the rest of the parameter list consists of the ordinary value parameters.

5.4.3 MethodReturnType

The `MethodReturnType` attribute records the type hint for the method return value.

```
MethodReturnType {
  u2 attribute_name_index;
  u4 attribute_length;
  TypeHint return_type;
}
```

For the `choose` method, the generated classfile contains:

```
public java.lang.Object choose(byte, byte, java.lang.Object, java.lang.Object);
  descriptor: (BBLjava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
  @MethodReturnType {
    return_type = M(0)
  }
```

The erased return descriptor is `Ljava/lang/Object;` however the source level return type is `T`, which is the first method type parameter. The attribute describes this relationship by recording a return type hint of `M(0)`.

The method attributes `MethodTypeparameterCount`, `MethodParameterType` and `MethodReturnType` together describe the generic relationships between the erased method descriptor and the source-level method signature. Note that the attributes describe the method declaration from the callee's point of view. For a method, a hint such as `K(0)` is interpreted relative to the receiver object. The reified VM can use the receiver object and class-level attributes to find the actual runtime reified type value for `K(0)`.

The compiler computes these attributes before erasure removes source-level generic information. A reified VM can use these attributes to execute the bytecode with the desired reified unboxed semantics, while a standard JVM can ignore them and execute the bytecode without reification.

5.5 Code-level Attributes

Method-level attributes describe declarations. They do not fully describe what happens inside a method body. Some type information belongs to individual bytecode instructions. For this reason, the compiler also emits code-level attributes. There are three code-level attributes in this design:

- `InvokeReturnType`, which describes the type hint of a particular invocation result;
- `BCNewTypeArgs`, which describes the reified type arguments associated with a particular `NEW` instruction;
- `ExtraBoxUnbox`, which marks Scala boxing and unboxing operations that can be ignored by the reified runtime.

Code-level attributes are necessary because the same method may be used in different type contexts. For example, a generic method may return `M(0)` from the callee's point of view, but `M(0)` may be instantiated to different types at different call sites. Consider the following example:

Source code

```
def id[T](t: T): T = t
def foo[U, V](u: U, v: V): Unit = {
  val x = id[U](u)
  val y = id[V](v)
}
val i42: Int = id[Int](42)
```

Reified code before backend

```
def id(reified$id$T: ReifiedValue)(t: Object): Object = t
def foo(reified$foo$U: ReifiedValue, reified$foo$V: ReifiedValue)(u: Object, v: Object): Unit = {
  val x = id(reified$foo$U)(u)
  val y = id(reified$foo$V)(v)
}
val i42: Int = id(IntTag)(42)
```

The value `i42` calls the generic method `id` with a concrete type `Int`. Whereas for `val x` and `y`, the call site is generic, and the return type of `id` is not concrete. For `x`, the return type is `U`, and for `y`, the return type is `V`. The code-level attributes can describe these differences with type hints at each call site.

One might question whether this kind of hint is necessary. Indeed, in terms of correctness, the reified VM should be able to determine whether the call site is generic or concrete by looking at the runtime reified type value. However, the point of the hint is to allow the reified VM to skip unnecessary work; a rule of thumb is that if the compiler can ascertain some useful information, it should emit that information as a hint, so that the runtime can use it directly.

5.5.1 InvokeReturnType

The `InvokeReturnType` attribute describes the type hint for the result of a particular invocation instruction. Each entry is associated with the bytecode offset of an invocation instruction.

```
InvokeReturnType {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 invoke_num;
  {
    u2 offset;
    TypeHint return_type;
  } invokes[invoke_num];
}
```

The offset identifies a particular invocation instruction in the method body. The return type hint describes the source-level return type of that instruction at the call. Consider the generated code for `id` and `foo` methods shown earlier.

```
public java.lang.Object id(byte, java.lang.Object){
   0: aload_2
   1: areturn
}
public void foo(byte, byte, java.lang.Object, java.lang.Object){
   0: aload_0
   1: iload_1
   2: aload_3
   3: invokevirtual #30 // Method id:(BLjava/lang/Object;)Ljava/lang/Object;
   6: astore        5   // x = id(reified$foo$U)(u)
   8: aload_0
   9: iload_2
  10: aload         4
  12: invokevirtual #30 // Method id:(BLjava/lang/Object;)Ljava/lang/Object;
  15: astore        6   // y = id(reified$foo$V)(v)
  17: return
}
public void <clinit>(){
  ...
  13: bipush        73    // IntTag
  15: bipush        42
  17: invokestatic  #26   // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
  20: invokevirtual #30   // Method id:(BLjava/lang/Object;)Ljava/lang/Object;
  23: invokestatic  #34   // Method scala/runtime/BoxesRunTime.unboxToInt:(Ljava/lang/Object;)I
  26: putstatic     #36   // Field i42:I
  29: return
}
```

The `id` method is generic, and its return type is `M(0)`. The two call sites of `id` in method `foo` are generic. For the return type assigned to `val x`, the return type is `M(0)`. For the return type assigned to `val y`, the return type is `M(1)`. The `M(0)` and `M(1)` are interpreted relative to the caller `foo`, which has two method type parameters. For the call site of `id` in method `<clinit>`, the return type is a concrete `IntTag`. The generated classfile contains:

```
...
public void foo(byte, byte, java.lang.Object, java.lang.Object){
   3: invokevirtual #30 // Method id:(BLjava/lang/Object;)Ljava/lang/Object;
                        // @InvokeReturnType { M(0) }
   6: astore        5   // x = id(reified$foo$U)(u)
   ...
  12: invokevirtual #30 // Method id:(BLjava/lang/Object;)Ljava/lang/Object;
                        // @InvokeReturnType { M(1) }
  15: astore        6   // y = id(reified$foo$V)(v)
}
public void <clinit>(){
  ...
```

```
13    20: invokevirtual #30   // Method id:(BLjava/lang/Object;)Ljava/lang/Object;
14                            // @InvokeReturnType { IntTag }
15  }
```

Whichever type hint the `InvokeReturnType` attribute assigns to the invocation, the reified VM interprets the hint relative to the caller's context. The distinction between `MethodReturnType` and `InvokeReturnType` is important. The former is attached to the callee method and describes the method declaration from the callee's point of view. This can be useful for the reified VM to decide what to put on top of the stack after the invocation. The latter is attached to the caller's code and describes the call site from the caller's point of view. This can be useful for the reified VM to decide how to interpret the value on top of the stack after the invocation.

5.5.2 BCNewTypeArgs

The `BCNewTypeArgs` attribute records the reified type arguments associated with object allocation sites. It contains one entry for each relevant `NEW` instruction.

```
BCNewTypeArgs{
  u2 attribute_name_index;
  u4 attribute_length;
  u2 new_num;
  {
    u2 offset;
    u2 class_layer_num;
    {
      u2 local_variable_count;
      u2 local_variable_index[local_variable_count];
    } class_reified_value[class_layer_num];
  } entries[new_num];
}
```

One might wonder: since the modified compiler is already passing reified type values into the constructor, why does the compiler also emit the reified locals and this attribute? The reason is that Java bytecode separates object allocation and constructor invocation into two instructions, `NEW` and a call to `<init>`. The reified VM needs to know the reified type values at the point of allocation (i.e, the `NEW` instruction) to determine the internal object layout and representation. The reified type values are passed into the constructor, to be stored as fields of the object, but they are not available at the point of allocation. Also, at the point of allocation, in order to determine the representation of the object, the

reified VM needs to know the reified type values for all class layers along the inheritance chain of the allocated object. The `BCNewTypeArgs` attribute provides this information to the reified VM.

`new_num` is the number of `NEW` instructions that have associated reified type arguments. Each entry contains the bytecode offset of the `NEW` instruction, the number of class layers that have reified type values, and for each class layer, a list of local variable indices that contain the reified type values for that class layer. Here, each class layer means the class on the inheritance chain of the allocated object. The local variable indices are the indices of the bytecode local variable table that contain the reified type values for the class type parameters of that class layer. The order of the class layer is from the least derived class, `Object`, to the most derived class, the allocated class. Consider the following example:

Source code

```
class Box[A](value: A)
class NamedBox[N, A](name: N,
  value: A) extends Box[A](value)
val b = new NamedBox[String,
↪  Double]("pi", 3.14)
```

Reified code before backend

```
... // the class definitions
val reified$NamedBox$N$Local: ReifiedValue =
↪  ReferenceTag
val reified$NamedBox$A$Local: ReifiedValue =
↪  DoubleTag
val reified$Box$A$Local: ReifiedValue =
↪  reified$NamedBox$A$Local
val b = new NamedBox(reified$NamedBox$N$Local,
↪  reified$NamedBox$A$Local)("pi", 3.14)
```

Generated bytecode

```
public void <clinit>(){
  ...
  10: bipush        76  // ReferenceTag
  12: istore_0          // reified$NamedBox$N$Local
  13: bipush        68  // DoubleTag
  15: istore_1          // reified$NamedBox$A$Local
  16: bipush        68  // DoubleTag
  18: istore_2          // reified$Box$A$Local
  19: new           #23 // class NamedBox
  22: dup
  23: iload_0           // reified$NamedBox$N$Local
  24: iload_1           // reified$NamedBox$A$Local
  25: ldc           #25 // String pi
  27: ldc2_w        #26 // double 3.14d
  30: invokestatic  #33 // Method scala/runtime/BoxesRunTime.boxToDouble:(D)Ljava/lang/Double;
  33: invokespecial #36 // Method NamedBox."<init>":(BBLjava/lang/Object;Ljava/lang/Object;)V
  36: putstatic     #38 // Field b:LNamedBox;
  ...
}
```

The bytecode contains a `NEW NamedBox` instruction. The instruction does not contain generic type arguments. The `BCNewTypeArgs` attribute supplies the missing reified type values.

```
public void <clinit>(){
  ...
  10: bipush        76  // ReferenceTag
  12: istore_0          // reified$NamedBox$N$Local
  13: bipush        68  // DoubleTag
  15: istore_1          // reified$NamedBox$A$Local
  16: bipush        68  // DoubleTag
  18: istore_2          // reified$Box$A$Local
  19: new           #23 // class NamedBox
      @BCNewTypeArgs {
        class_layer_num = 2
        {
          local_variable_count = 1
          { 2 // local slot 2: reified$Box$A$Local
          }
        },
        {
          local_variable_count = 2
          { 0 // local slot 0: reified$NamedBox$N$Local
            1 // local slot 1: reified$NamedBox$A$Local
          }
        }
      }
  ...
}
```

5.5.3 ExtraBoxUnbox

The `ExtraBoxUnbox` attribute marks Scala boxing and unboxing operations that are needed for standard JVM execution but not necessary in the reified VM. Consider again the following Scala program along with its generated reified bytecode:

Source code

```
def id[T](t: T): T = t
val x: Int = id[Int](42)
```

Reified code before backend

```
def id(reified$id$T: ReifiedValue)(t: Object): Object
↪  = t
val x: Int = id(IntTag)(42)
```

Generated Reified Bytecode

```
1  public void <clinit>(){
2    ...
3    13: bipush        73
4    15: bipush        42
5    17: invokestatic  #26  // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
6    20: invokevirtual #30  // Method id:(BLjava/lang/Object;)Ljava/lang/Object;
7    23: invokestatic  #34  // Method scala/runtime/BoxesRunTime.unboxToInt:(Ljava/lang/Object;)I
8    26: putstatic     #36  // Field x:I
9    ...
10 }
```

The boxing and unboxing calls are required by the standard JVM due to type erasure; method `id` takes in and returns a `java.lang.Object`. However, the source-level call is instantiated with `Int`, and the reified type value `IntTag` along with method level type hints on `id` made this information explicit to the reified VM. In the reified unboxed semantics, the argument and result can be treated as primitive `Int` values.

The `ExtraBoxUnbox` attribute identifies boxing and unboxing operations that arise from this transition from erased bytecode semantics to reified unboxed semantics:

```
ExtraBoxUnbox {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 count;
  u2 offsets[count];
}
```

This attribute does not remove the boxing and unboxing operations from the bytecode. The boxing and unboxing remains there so that the program can execute according to ordinary JVM semantics. Instead, the attribute gives the reified VM permission to ignore these calls. Thus, the reified VM can execute the program with the desired reified unboxed semantics.

5.6 End-to-End Example

This section shows how the attributes work together on the running example from Section 3.3.

Source code

```
class Box[A](value: A) {
  def get: A = value
}
def makeBox[T](x: T):
↪  Box[T] =
  new Box[T](x)
val b = makeBox[Int](42)
```

Reified code before backend

```
class Box(reified$Box$A: ReifiedValue)(value: Object){
  def get(): Object = value
}
def makeBox(reified$makeBox$T: ReifiedValue)(x: Object):
↪  Box = {
  val reified$Box$A$Local: ReifiedValue =
  ↪  reified$makeBox$T
  new Box(reified$Box$A$Local)(x)
}
val b: Box = makeBox(IntTag)(42)
```

Figure 5.4 shows the generated reified bytecode with attributes for the running example.

The generated class `Box` has one class-level type parameter, therefore the compiler emits the following class-level attributes:

```
@ClassTypeParameterCount {
  count = 1
}
@ClassTypeParamList {
  class_type_param_num = 1
  0 -> Fieldref Box.reified$Box$A:B
}
```

The field `value` has source-level type `A`, which is the first class type parameter. The compiler emits the following field-level attribute to record the source-level meaning of the field:

```
private final java.lang.Object value;
  descriptor: Ljava/lang/Object;
  @FieldType(K(0))
```

The method `get` returns a value of source-level type `A`. Its erased JVM descriptor returns `java.lang.Object`, but the method-level return attribute records the source-level meaning:

```
@MethodReturnType(K(0))
Object get() { ... }
```

The method `makeBox` has one method type parameter `T`. The first ordinary parameter in its parameter list has source-level type `T`. The method-level attributes therefore describe these relations as:

```
@MethodTypeParameterCount {
  count = 1
}
@MethodParameterType {
  parameter_num = 2
  0 -> Byte
  1 -> M(0)
}
```

In `<clinit>`, the call site of `makeBox` is passed a boxed int, but in the reified unboxed semantics, the passed parameter is a primitive `Int`. So the compiler emits an `ExtraBoxUnbox` attribute to mark the boxing operation at offset 17 as unnecessary for the reified VM.

```
@ExtraBoxUnbox {
  1,
  offsets = {17}
}
```

```
1  public class Box {
2    // @ClassTypeParameterCount { count = 1 }
3    // @ClassTypeParamList { class_type_param_num = 1,
4    //       0 -> Fieldref Box.reified$Box$A:B }
5    public final byte reifiedField$Box$A;
6    private final Object value; // @FieldType(K(0))
7    public Box(byte, Object);
8        0: aload_0
9        1: iload_1
10       2: putfield      #16  // Field reifiedField$Box$A:B
11       5: aload_0
12       6: aload_2
13       7: putfield      #18  // Field value:Ljava/lang/Object;
14      10: aload_0
15      11: invokespecial #21  // Method java/lang/Object."<init>":()V
16      14: return
17   // @MethodReturnType(K(0))
18   public Object get();
19       0: aload_0
20       1: getfield      #18  // Field value:Ljava/lang/Object;
21       4: areturn
22 }
23 class foo{
24   static final Box b;
25   public static void <clinit>();
26       ...
27       13: bipush        73   // IntTag
28       15: bipush        42
29       17: invokestatic  #27  // Method scala/runtime/BoxesRunTime.boxToInteger:(I)Ljava/lang/Integer;
30       // @ExtraBoxUnbox
31       20: invokevirtual #31  // Method makeBox:(BLjava/lang/Object;)LBox;
32       23: putstatic     #33  // Field b:LBox;
33       26: return
34   // @MethodTypeParameterCount { count = 1 }
35   // @MethodParameterType { parameter_num = 2,
36   //      0 -> Byte
37   //      1 -> M(0) }
38   public Object Box makeBox(byte, Object);
39       0: iload_1
40       1: istore_3
41       2: new           #45  // class Box
42                         // @BCNewTypeArgs({local slot 3})
43       5: dup
44       6: iload_3
45       7: aload_2
46       8: invokespecial #48  // Method Box."<init>":(BLjava/lang/Object;)V
47      11: areturn
48 }
```

Figure 5.4: Bytecode with attributes for the running example. The attributes are shown as comments in the bytecode.

Chapter 6

Evaluation

The preceding chapters described the design and implementation of the modified Scala compiler. This chapter evaluates whether reification can be integrated into the production Scala deeply enough to support realistic Scala programs. This is a different question from the one usually considered by work on generic specialization, which typically focuses on whether a piece of code can be made faster.

This distinction is important when comparing this work with TASTyTruffle. TASTyTruffle demonstrates that reified type information can enable efficient representations and dynamic specialization, but it implements a core subset of Scala and evaluates that subset using small benchmarks. These restrictions are reasonable for evaluating the performance potential of reified generics, but they prevent the system from directly executing much of ordinary Scala code and the Scala standard library.

The evaluation considers several aspects of the implementation. First, it evaluates whether the modified compiler produces the required type information and whether that information survives to execution. Second, it evaluates whether the type information introduced by the compiler is consumed by the reified VM for it to make type decisions. Third, it evaluates whether the compiler transformation scales from small examples to realistic Scala code, including the standard library and benchmarks on the standard library.

6.1 End-to-End Mini-Benchmarks

Before evaluating the compiler on the Scala standard library, we use some custom mini-benchmarks to test whether reified type information is propagated correctly through complete executable programs. Each benchmark repeatedly performs a computation and checks its result against a precomputed expected value. A benchmark reports an error if the computed result is incorrect. All benchmarks compile successfully using the modified compiler and execute to completion without reporting errors. Their complete source code is included in Appendix A.

Benchmark	Supporting classes	Features exercised

`ArraySum`	`ArrayContainer[T]`
	`BinaryOp[T]`
	`IntAddition`
		Generic arrays
		Generic method parameters and return values
		Inheritance from a generic abstract class
		Instantiation of a generic abstraction with `Int`

`HeapSort`	`PriorityQueue[A]` `Vector[A]`	Array-backed generic collections Generic reads and writes Repeated insertion and removal of primitive values in generic collections
`SlidingWindow`	`MaxQueue[A]` `Deque[A]`	Generic inheritance Inherited generic array-backed storage Generic method calls through a subclass Use of a generic queue instantiated with `Int`
`HashMap`	`HashMap[T]`	Generic array Generic method parameters and return values Repeated storage and retrieval using `HashMap[Int]`

Table 6.1: Custom mini-benchmarks used for end-to-end evaluation.

Table 6.1 summarizes the benchmarks and the compiler reified features that they exercise. These benchmarks are created to somewhat resemble the generic pattern to be used when writing generic Scala code. These benchmarks are still considerably smaller than the Scala standard library, but they provide a useful intermediate evaluation level. Each benchmark contains enough interacting generic code that successful execution requires reified type values and type hints to survive multiple compiler phases. For example, the benchmarks combine generic arrays with inheritance, virtual dispatch, mutable fields, and primitive instantiations.

6.1.1 Use of Reified Information by the VM

Successful execution alone does not mean that the reified VM actually uses the information generated by the compiler. In order to measure this directly, the reified VM is instrumented to count operations for which an erased reference representation is replaced by a primitive representation using the reified type information.

The instrumentation records six categories of runtime decisions. The reified VM records its decision in handling erased method parameters. A generic method may have the JVM descriptor `Object` for a parameter when its runtime type parameter indicates that it is an `int`. When the corresponding reified information identifies such occasions, the counter `Parameter` is incremented by one. Similarly, the counter `Local` is increased for such decisions for JVM instructions `ALOAD`, `ASTORE`; the counter `Return` is increased for decisions for `ARETURN`. The VM also records its decisions in the counter `Field` when handling field instructions `GETFIELD` and `PUTFIELD`. The counter `Array` counts the number of decisions made on array operations (`array_apply` and `array_update`). These instructions all previously operated on references; these references may be erased generic values. Thus, the reified VM interprets their values with type hints and runtime reified type values. The counter `Ignored (un)Box` counts the number of invocations of runtime boxing/unboxing helpers that are ignored by

Benchmark	Parameter	Local	Return	Field	Array	Ignored (un)Box
ArraySum	1.01B	2.02B	505.5M	0	505.5M	1.52B
HeapSort	371.8M	583.9M	8.27M	0	750.6M	371.8M
SlidingWindow	185.8M	269.2M	123.4M	0	124.0M	185.2M
HashMap	105.0M	52.5M	52.5M	0	105.0M	157.4M

Table 6.2: Runtime representation decisions made by the reified VM for the mini-benchmarks

Benchmark	Baseline (ms)	Reified (ms)	Speedup
ArraySum	662.842	67.434	9.83×
HeapSort	2155.458	146.881	14.67×
HashMap	93.806	48.049	1.95×
SlidingWindow	17711.295	35.614	497.31×

Table 6.3: Execution time for the mini benchmarks. Lower execution time is better.

the VM.

The table 6.2 shows that the reified information is used throughout execution for the four mini-benchmarks. The numbers in the table show that the reified information is used throughout execution rather than merely existing in the classfiles. For example, the `ArraySum` and `HeapSort` benchmarks cause the reified VM to make approximately 500 million and 750 million representation decisions, respectively, for array operations. This is expected, as both benchmarks perform a large number of array accesses and updates. These measurements provide direct evidence that the information emitted by the modified Scala compiler reaches the reified VM and affects its execution decisions.

6.1.2 Performance

The execution time of the mini-benchmarks is compared against two configurations. The baseline uses the unmodified Scala compiler and the unmodified Espresso, while the reified configuration uses the modified Scala compiler and the reified Espresso implementation. The two sources are the same in both configurations.

Since the JVM needs to warm up and perform runtime compilation, the benchmarks are run for 10 rounds. The table 6.3 reports the median execution time of the final five rounds for each configuration.

From the table, the reified configuration is faster on all four benchmarks. These metrics demonstrate that the additional type information is not merely descriptive metadata: the reified VM can actually use it to change the representation and execution of generic values, and this change can boost the performance of generic code.

It is worth noting that the measurements are done on four micro benchmarks, and should not be interpreted as a general performance comparison between ordinary Scala and reified Scala. A full performance comparison and explanations of how such performance gains are possible are outside of scope for this thesis.

6.2 Scaling to the Scala Standard Library

The main scalability experiment recompiles the Scala standard library using the modified compiler. This is a substantially stronger test than compiling a collection of isolated generic examples.

The standard library contains a dense combination of the features that stress the implementation:

- large generic trait hierarchy patterns [12];
- default methods and mixin-generated forwarders;
- higher-order methods and closure generation;
- nested, local, anonymous, and module classes;
- generic arrays and primitive array operations;
- Java interfaces and Java library calls; and
- compiler-generated bridges and adaptation methods.

Compiling the library therefore tests whether the reification transformation composes with the production compiler pipeline at scale. A local transformation that works only for direct calls to handwritten generic methods would fail early on this workload.

The same library sources are compiled by the production Scala compiler and the modified compiler. The modified compiler successfully compiles the standard library. The build processes 651 Scala sources and 57 Java sources and emits 5055 classfiles. The regular Scala compiler emits 3663 classfiles. The difference is due to the additional annonymous classes generated by the reification transformation for closures.

A successful build provides evidence that the reification transformation can be applied while compiling the Scala standard library, rather than only to small self-contained examples. Moreover, the Scala repository provides a test suite that can be run on the built library. In the suite, there are 2058 tests in total, of which 25 fail and 2033 pass.

The failing tests all fall into two categories: JVM serialization tests and Scala allocation size tests. The serialization tests fail because the JVM requires that a serializable class

needs an accessible zero-arg constructor on its first non-serializable superclass. The reification transformation does not currently generate such a constructor, so the serialization tests fail. The allocation size tests fail because the reification transformation generates additional parameters, fields, and methods, which increases the size of some classes. The allocation size tests check that certain classes are below a size threshold, so they fail when the reification transformation increases the class size.

The tests are run on some generic programs, including:

- `List[Int]` and `List[String]` operations;
- `Option`, `Either`, and tuples;
- `Iterator.map`, `filter`, `foldLeft`, and `reduce`;
- mutable and immutable maps;
- sorting through `Ordering[T]`;
- numeric operations through `Numeric[T]`; and
- primitive and reference arrays.

The remaining 2033 tests pass, which demonstrates that the reification transformation does not break the execution of the standard library.

6.2.1 Standard Library Benchmarks

Compiling and testing the standard library demonstrates that the reification transformation scales to a large Scala codebase, but these results alone do not show that the reified information generated for standard-library code reaches the reified VM or influences its decision for value representation. To evaluate this directly, the four mini-benchmarks are also adapted to use generic data structures from the reified build of the Scala standard library.

The original mini-benchmarks used simplified generic data structures that mimic the standard library counterparts; in these benchmarks, the corresponding standard library implementations are used instead.

For example, `HeapSort` uses `collection.mutable.PriorityQueue[A]`, `SlidingWindow` uses `collection.mutable.ArrayDeque[A]`, and `HashMap` uses `collection.mutable.HashMap[K,V]`.

The reified VM is instrumented in the same way as for the mini-benchmarks in the previous section. Table 6.4 reports the runtime type decisions made by the reified VM while executing the benchmarks against the reified standard library.

Benchmark	Parameter	Local	Return	Field	Array	Ignored (un)Box
ArraySum	1.01B	3.01B	1.01B	0	4.5K	2.01B
HeapSort	173.4M	197.7M	22.5K	0	106.8M	41.3M
HashMap	775.0M	1.24B	308.5M	361.1M	206.7M	18.9K
SlidingWindow	1.42B	3.05B	4.43B	0	1.82B	1.01B

Table 6.4: Runtime representation decisions for the standard-library monomorphic benchmarks

The results show that representation decisions continue to occur when the benchmarks execute through standard library data structures rather than the simplified data structures used by the original mini-benchmarks. For example, the `HashMap` benchmark causes around 360 million representation decisions for generic field accesses. These decisions occur since the standard library `HashMap` implementation stores generic keys and values in internal generic node structures, a `Node[K, V]` class, each with a generic key and value.

These results provide direct runtime evidence that the type information generated by the modified compiler is propagated through the Scala standard library and is actively used by the reified VM to determine the representation of erased generic values.

6.3 Comparison with TASTyTruffle

TASTyTruffle and this work both aim to make generic type information available at runtime, but they do so at different levels of the Scala implementation and therefore evaluate different claims.

TASTyTruffle interprets TASTY directly using a research interpreter built on Truffle. Because TASTY retains substantially more source-level type information than ordinary JVM bytecode, TASTyTruffle can use precise generic type information during execution and specialize generic data representations dynamically. Its evaluation demonstrates the performance potential of this approach. However, the interpreter implements only a core subset of Scala. It supports features such as primitive values, classes with single inheritance, singleton objects, conditionals, loops, early returns, and method dispatch, while features such as multiple inheritance through traits, pattern matching, and Java interoperability are outside its stated scope [5]. Its evaluation is therefore based on a small collection of generic benchmarks rather than on complete ordinary Scala applications or the Scala standard library.

This work instead modifies the production Scala 3 compiler and its JVM backend. Most importantly, being able to recompile the Scala standard library demonstrates that the transformation applies to a large body of production generic code rather than only to a restricted language subset or a set of microbenchmarks. Source programs continue to pass through the existing Scala frontend, type checker, desugarings, and lowering phases. The

compiler inserts reified type values before erasure, preserves selected generic type information as type hints, and emits ordinary JVM bytecode augmented with custom classfile attributes. As a result, the implementation does not need to reproduce Scala language semantics in a separate interpreter. Indeed, the reified VM will have to implement the reified unboxed semantics in order to execute the emitted bytecode. But the implementation is not as complex as implementing a full production interpreter.

Chapter 7

Related Work

This thesis is related to work on implementing generics, specializing generic programs, preserving type information at runtime, and optimizing Scala programs on VMs. This chapter discusses previous work in these areas.

7.1 Implementation of Generics

Implementations of generics can be broadly classified according to how they represent values and generate code for different type arguments. Morrison et al. describe uniform, textual, and tagged implementations of polymorphic abstractions [13]. In the terminology used throughout this thesis, these categories correspond to different implementation strategies for generic classes and methods.

In a uniform implementation, all instantiations of a generic definition use a common representation and a single implementation. Type erasure, as used by Java and the standard Scala compiler, is an example of this approach. An erased generic value is generally represented as an object reference, regardless of its source-level type. With uniform representation, primitive values must be converted to and from objects when they pass through generic contexts.

A textual implementation generates a distinct version of a generic definition for each relevant combination of type arguments. This approach is also known as monomorphization. Because the generated version operates on concrete types, it can use specialized fields, local variables, instructions, and calling conventions. The disadvantage is that multiple instantiations may produce multiple copies of nearly identical code.

A tagged implementation uses runtime type information to select the representation of generic values. Generic definitions can retain a single implementation, but that implementation receives or otherwise accesses tags that describe its runtime type arguments. Intensional type analysis provides a formal basis for compiling polymorphic programs in which code can inspect runtime representations of types [8, 3].

The design in this thesis is closest to tagged polymorphism. A generic code receives lightweight reified type values for the type parameters.

7.2 Static Specialization on the JVM

Since the standard JVM erases generic type arguments, several systems attempt to recover efficient representations by generating specialized code before execution.

Scala Specialization. Dragos and Odersky introduced user-directed specialization in the Scala 2 compiler [4]. A programmer can annotate a type parameter with `@specialized`, causing the compiler to generate versions of the associated classes and methods for selected primitive types. The specialized versions use primitive JVM descriptors and primitive bytecode instructions, avoiding boxing in the specialized parts of the program.

Scala specialization can provide efficient representations without requiring modifications to the JVM. Its principal cost is code duplication. A definition specialized for several primitive types can produce many generated classes, methods, bridges, and forwarding methods. The number of versions can grow further when a definition has multiple specialized type parameters. Specialization is also programmer-directed: generic code for which specialization was not requested remains erased.

The system in this thesis emits one transformed generic definition, supplies the definition with reified type values, and is executed on a reified VM. This avoids the systematic code-size growth of full static specialization. The tradeoff is that the generated bytecode depends on a reified VM to obtain the reified unboxed semantics; an ordinary JVM continues to interpret the underlying erased descriptors and boxing operations according to standard JVM semantics.

Miniboxing. Miniboxing was developed to improve the tradeoff between execution speed and generated code size in Scala specialization [19]. Rather than generating a version for every primitive representation, all primitive types are encoded using `Long`, while reference types are encoded using `Object`. Runtime tags distinguish the primitive type stored in the carrier, and conversion operations translate between the carrier and the concrete primitive representation.

Compared with ordinary specialization, miniboxing can reduce code bloat, since it generates fewer specialized versions. However, code bloat is still a concern, since the number of specialized versions still grows exponentially with the number of specialized type parameters. Like specialization, there is also the concern that it needs boxing and unboxing operations at compatibility boundaries, so it does not eliminate boxing globally. What is more, since primitive values are encoded using a common representation `Long`, they are not always compactly represented. The paper notes that if the type argument is

`Byte`, the miniboxed representation can have an eight-times larger memory footprint than monomorphic code that uses `Byte` directly [19].

Miniboxing and the approach in this thesis both associate generic values with compact runtime type information. They differ in the role of the uniform representation. Miniboxing selects a small set of concrete JVM carrier representations and compiles generic operations around those carriers. In this thesis, the inserted reified type value identifies the desired representation.

Miniboxing is primarily a compiler-side translation that remains executable on a standard JVM. This thesis divides responsibility between the Scala compiler and a modified VM. The compiler performs propagation and emits metadata, while the VM is responsible for exploiting that metadata to reified unboxed semantics execution.

7.2.1 Specializing Java Programs

NextGen proposed an alternative translation for Java generics that preserves runtime type information while remaining compatible with existing Java classfiles [2]. NextGen generates specialized wrapper classes that make generic type arguments available for operations such as runtime type tests. Its main objective is to support operations that Java's erased generics cannot express. It does not provide a uniform solution for instantiating generic definitions with primitive types.

JCS specializes selected Java collection classes ahead of normal Java compilation [7]. It generates concrete variants of generic data structures for particular type arguments, allowing their fields and operations to use more precise representations. Like Scala specialization, this approach produces conventional JVM bytecode by generating additional program definitions.

7.3 Reified Generics in Managed Runtimes

A managed runtime can implement generic types more directly when its intermediate representation and execution model retain generic type arguments.

The .NET Common Language Runtime. The .NET Common Language Runtime (CLR) provides native support for generic classes and methods [9] for the C# programming language. Generic type information is preserved in Common Intermediate Language metadata, allowing the runtime to share implementations for compatible reference types while using specialized representations for value types.

This thesis follows a similar goal, but targets the JVM, where generic type arguments are normally erased. Unlike the CLR, the proposed system does not preserve complete runtime generic types.

Project Valhalla. Project Valhalla investigates extensions to Java and the JVM that reduce the distinction between primitive values and objects and permit more specialized representations for generic code [15]. It aims to introduce platform-level support for generic specialization across the Java language and JVM ecosystem.

Valhalla and this thesis share the observation that conventional erased generic bytecode does not provide enough information for specialized runtime representations. They differ in scope. Valhalla is a platform-wide evolution of the Java language. This thesis develops a reification system within the existing JVM framework. The approach in this thesis can therefore be viewed as an experimental investigation of reified generic execution for Scala rather than an alternative JVM standard.

7.4 Reifying Types in Scala

Scala already contains mechanisms for carrying selected type information into erased code. Other Scala implementations have also explored other reification strategies.

Runtime Type Evidence. As discussed in Chapter 2, `ClassTag` values allow Scala programs to request the erased runtime class associated with a type parameter. However, `ClassTag` is implemented through Java reflection, which is considered slow and heavyweight.

The reified type values in this thesis, however, are designed to be lightweight and efficient. They are represented as JVM `byte` values, which can be passed through the program without the overhead of reflection. This allows the reified type information to be used in performance-critical code paths.

Reifying Scala Types as JVM Objects. Schinz investigated a Scala compilation strategy in which Scala types are reified as JVM objects [17]. Runtime type objects allow operations such as type tests and pattern matching to retain information that would otherwise be lost through erasure. Reifying the type system at the language level also introduces runtime and memory costs.

The current thesis deliberately uses a less expressive representation. Its byte-valued tags cannot describe arbitrary Scala types or support general runtime type inspection. They preserve only distinctions that affect the representation of generic values. This restricted encoding reduces the amount of runtime metadata and also gives the reified VM just enough information to use a compact representation.

TASTyTruffle. TASTyTruffle is the most direct predecessor of this work [5]. TASTyTruffle demonstrates that retained Scala type information can improve the execution of generic programs. Its main limitation is language and standard library coverage.

Chapter 8

Conclusion

This thesis followed the goal of TASTyTruffle in improving the execution of generic types in Scala by preserving representation-relevant type information through reification. It presents a reified unboxed execution model for generic types in Scala. It introduces transformations to the Scala 3 compiler that preserve representation-relevant generic type information after erasure and communicates it to a reified VM. The compiler inserts lightweight runtime reified type values for method and class type parameters and emits custom JVM classfile attributes that describe how erased parameters, return values, fields, method invocations, and object allocations relate to those values.

Implementing this design required integrating reification with the existing Scala compiler pipeline. The implementation preserves type information before erasure and propagates reified type values through generic methods, classes, constructors, traits, inheritance, nested classes, closures, generic arrays, bridges, and other compiler-generated constructs. The resulting classfiles retain ordinary JVM bytecode while providing additional information that a reified VM can use to recover the reified unboxed semantics.

The evaluation demonstrates that the approach scales beyond small examples. In addition to compiling and executing the custom mini-benchmarks, the modified compiler successfully compiled the Scala standard library, processing hundreds of Scala and Java source files and producing more than five thousand classfiles. Of all the standard library tests, 98.7% passed. The remaining failures were associated mainly with expected structural changes introduced by reification, such as modified constructor signatures and increased object sizes. These results indicate that the transformation is sufficiently integrated with the production compiler to support a large and realistic Scala codebase.

The work does not yet establish the performance benefits of reified execution. The current evaluation focuses on compiler correctness, language-feature coverage, and scalability.

Future work includes completing the integration with the modified Espresso VM, evaluating the effect of reification on boxing, memory allocation, and execution performance, and extending support to additional Scala language features. The reification system could

also be extended to support additional type distinctions, such as higher-kinded types and type members.

Overall, this thesis demonstrates that representation-relevant generic type information can be preserved through the production Scala 3 compiler and exposed at the JVM bytecode level without relying on static specialization. The successful compilation of the Scala standard library provides evidence that this approach can scale to realistic Scala programs and forms a practical basis for future reified-generic optimizations in a JVM implementation.

References

[1] Marat Akhin, Mikhail Belyaev, et al. *Kotlin Language Specification: Kotlin/Core*. JetBrains and JetBrains Research, 2020.

[2] Robert Cartwright and Guy L. Steele. Compatible genericity with run-time types for the java programming language. *SIGPLAN Not.*, 33(10):201–215, October 1998.

[3] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *SIGPLAN Not.*, 34(1):301–312, September 1998.

[4] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, page 42–47, New York, NY, USA, 2009. Association for Computing Machinery.

[5] Matt D'Souza, James You, Ondřej Lhoták, and Aleksandar Prokopec. Tastytruffle: Just-in-time specialization of parametric polymorphism. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):1561–1588, 2023.

[6] Ekaterina Goltsova, editor. *Optimizing Java on Truffle*. 2022.

[7] Dan Graur, Rodrigo Bruno, and Gustavo Alonso. Specializing generic java data structures. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2021, page 45–53, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 130–141, New York, NY, USA, 1995. Association for Computing Machinery.

[9] Andrew Kennedy and Don Syme. Design and implementation of generics for the .net common language runtime. *SIGPLAN Not.*, 36(5):1–12, May 2001.

[10] LAMP/EPFL. Internals. https://scala-lang.org, 2026. Accessed: 2026-07-11.

[11] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Virtual Machine Specification*. Oracle America, Inc., java se 17 edition, 2021.

[12] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. *SIGPLAN Not.*, 43(10):423–438, October 2008.

[13] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Trans. Program. Lang. Syst.*, 13(3):342–371, July 1991.

[14] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. 2004.

[15] OpenJDK. Project valhalla. https://openjdk.org/projects/valhalla/. Accessed July 2026.

[16] Oracle. Autoboxing and unboxing, 2026. Accessed: July 27, 2026.

[17] Michel Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, École polytechnique fédérale de Lausanne, 2005.

[18] Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1):11–49, 2000.

[19] Vlad Ureche, Cristian Talau, and Martin Odersky. Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. *SIGPLAN Not.*, 48(10):73–92, October 2013.

Appendix A

Additional Scala Code

```scala
class ArrayContainer[T](val storage: Array[T]) {
  def reduce(fn: BinaryOp[T]): T = {
    var state = storage(0)
    var i = 1
    val length = storage.length
    while (i < length) {
      state = fn.apply(state, storage(i))
      i += 1
    }
    state
  }
}
```

Figure A.1: `ArrayContainer` class

```scala
abstract class BinaryOp[T] {
  def apply(lhs: T, rhs: T): T
}

class IntAddition extends BinaryOp[Int] {
  override def apply(lhs: Int, rhs: Int): Int = lhs + rhs
}
```

Figure A.2: `BinaryOp` and `IntAddition` classes

```scala
class Deque[A](initialCapacity: Int):
  private var elements = new Array[A](math.max(1, initialCapacity))
  private var first = 0
  private var currentSize = 0
  def isEmpty: Boolean = currentSize == 0
  def nonEmpty: Boolean = currentSize != 0
  def size: Int = currentSize
  def pushFront(value: A): Unit =
    ensureCapacity(currentSize + 1)
    first = wrap(first - 1)
    elements(first) = value
    currentSize += 1
  def pushBack(value: A): Unit =
    ensureCapacity(currentSize + 1)
    elements(physicalIndex(currentSize)) = value
    currentSize += 1
  def front: A = elements(first)
  def back: A = elements(physicalIndex(currentSize - 1))
  def popFront(): A =
    val result = elements(first)
    first = wrap(first + 1)
    currentSize -= 1
    if currentSize == 0 then first = 0
    result
  def popBack(): A =
    val index = physicalIndex(currentSize - 1)
    val result = elements(index)
    currentSize -= 1
    if currentSize == 0 then first = 0
    result
  def apply(index: Int): A = elements(physicalIndex(index))
  def update(index: Int, value: A): Unit = elements(physicalIndex(index)) = value
  def clear(): Unit =
    first = 0
    currentSize = 0
  private def physicalIndex(logicalIndex: Int): Int = wrap(first + logicalIndex)
  private def wrap(index: Int): Int =
    val capacity = elements.length
    if index >= capacity then index - capacity
    else if index < 0 then index + capacity
    else index
  private def ensureCapacity(requiredCapacity: Int): Unit =
    if requiredCapacity > elements.length then
      var newCapacity = elements.length
      while newCapacity < requiredCapacity do
        newCapacity *= 2
      val resized = new Array[A](newCapacity)
      var index = 0
      while index < currentSize do
        resized(index) = elements(physicalIndex(index))
        index += 1
      elements = resized
      first = 0
```

Figure A.3: `Deque` class

```scala
final class HashMap[T](val capacity: Int):
  private val keys = new Array[Int](capacity)
  private val values = new Array[T](capacity)
  def get(key: Int, defaultValue: T): T =
    val index = findSlot(key)
    if keys(index) == key then values(index) else defaultValue
  def set(key: Int, value: T): Unit =
    val index = findSlot(key)
    keys(index) = key
    values(index) = value
  private def findSlot(key: Int): Int =
    var index = key % capacity
    while keys(index) > 0 && keys(index) != key do
      index += 1
      if index == capacity then index = 0
      println(s"findSlot: key=$key, index=$index, keys=${keys.mkString(",")}")
    index
```

Figure A.4: `HashMap` class

```scala
final class MaxQueue[A](initialCapacity: Int)(using ordering: Ordering[A]) extends
↪  Deque[A](initialCapacity):
  def enqueue(value: A): Unit =
    while nonEmpty && ordering.lt(back, value) do
      popBack()
    pushBack(value)

  def dequeue(value: A): Unit =
    if nonEmpty && ordering.equiv(front, value) then
      popFront()

  def getMax: A = front

```

Figure A.5: `MaxQueue` class

```scala
import scala.annotation.tailrec
final class PriorityQueue[A](using ordering: Ordering[A]):
  private val heap = bcGen.Performance.Vector.empty[A]
  def isEmpty: Boolean = heap.isEmpty
  def nonEmpty: Boolean = heap.nonEmpty
  def size: Int = heap.size
  def enqueue(value: A): Unit =
    heap.pushBack(value)
    siftUp(heap.size - 1)
  def peek: A = heap(0)
  def dequeue(): A =
    val result = heap(0)
    val last = heap(heap.size - 1)
    heap.popBack()
    if heap.nonEmpty then
      heap(0) = last
      siftDown(0)
    result
  def clear(): Unit = heap.clear()
  @tailrec
  private def siftUp(startIndex: Int): Unit =
    if startIndex > 0 then
      val parent = (startIndex - 1) / 2
      if ordering.lt(heap(parent), heap(startIndex)) then
        swap(parent, startIndex)
        siftUp(parent)
  @tailrec
  private def siftDown(parent: Int): Unit =
    val left = parent * 2 + 1
    val right = left + 1
    if left < heap.size then
      val largest =
        if right < heap.size && ordering.gt(heap(right), heap(left)) then right
        else left
      if ordering.lt(heap(parent), heap(largest)) then
        swap(parent, largest)
        siftDown(largest)
  private def swap(i: Int, j: Int): Unit =
    val tmp = heap(i)
    heap(i) = heap(j)
    heap(j) = tmp
```

Figure A.6: `PriorityQueue` class

```scala
final class Vector[A] private (initialCapacity: Int):
  private var elements = new Array[A](math.max(1, initialCapacity))
  private var currentSize = 0
  def isEmpty: Boolean = currentSize == 0
  def nonEmpty: Boolean = currentSize != 0
  def size: Int = currentSize
  def pushBack(value: A): Unit =
    ensureCapacity(currentSize + 1)
    elements(currentSize) = value
    currentSize += 1
  def popBack(): Unit = currentSize -= 1
  def apply(index: Int): A = elements(index)
  def update(index: Int, value: A): Unit = elements(index) = value
  def clear(): Unit = currentSize = 0
  private def ensureCapacity(requiredCapacity: Int): Unit =
    if requiredCapacity > elements.length then
      var newCapacity = elements.length
      while newCapacity < requiredCapacity do
        newCapacity *= 2

      val resized = new Array[A](newCapacity)
      var index = 0
      while index < currentSize do
        resized(index) = elements(index)
        index += 1
      elements = resized
object Vector:
  def empty[A]: Vector[A] = new Vector[A](1)
```

Figure A.7: `Vector` class

```scala
object ArraySum {
  private final val T = 100
  private final val OUTER_REPEAT = 100
  private final val INNER_REPEAT = 1000
  private final val N = 1000

  private def benchmark(): Unit = {
    val storage = Array.range(0, N)
    val arr = new ArrayContainer[Int](storage)
    val adder = new IntAddition
    var s = 0
    var i = 0
    while (i < INNER_REPEAT) {
      s += arr.reduce(adder)
      i += 1
    }
    if (s != 499500000) {
      println(s"Error: s=$s")
    }
  }

  @main def runArraySum(): Unit = {
    var t = 1
    while (t <= T) {
      val startTime = System.nanoTime()
      var r = 0
      while (r < OUTER_REPEAT) {
        benchmark()
        r += 1
      }
      val duration = System.nanoTime() - startTime
      println(s"round $t: ${duration / 1000}us")
      t += 1
    }
  }
}
```

Figure A.8: `ArraySum` benchmark

```scala
object HashMap:
  private final val T = 100
  private final val OUTER_REPEAT = 100
  private final val INNER_REPEAT = 100
  private final val N = 1024
  private final val CAPACITY = 8192
  private val keys =
    val result = new Array[Int](N)
    var i = 0
    while i < N do
      result(i) = i * 12345 + 123
      i += 1
    result
  private val expectedSum =
    var result = 0
    var i = 0
    while i < N do
      result += i * 3 + 7
      i += 1
    result
  private def benchmark(): Unit =
    var repeat = 0
    val map = new HashMap[Int](CAPACITY)
    while repeat < INNER_REPEAT do
      var i = 0
      while i < N do
        map.set(keys(i), i * 3 + 7)
        i += 1
      var total = 0
      i = N - 1
      while i >= 0 do
        total += map.get(keys(i), -1)
        i -= 1
      if total != expectedSum then
        println(s"Error: total=$total, expectedSum=$expectedSum")
      repeat += 1
  @main def runHashMap(): Unit =
    var t = 1
    while t <= T do
      val startTime = System.nanoTime()
      var r = 0
      while r < OUTER_REPEAT do
        benchmark()
        r += 1
      val duration = System.nanoTime() - startTime
      println(s"round $t: ${duration / 1000}us")
      t += 1
```

Figure A.9: `HashMap` benchmark

```scala
object HeapSort:
  private final val T = 100
  private final val OUTER_REPEAT = 100
  private final val INNER_REPEAT = 4
  private final val N = 4096
  private val values =
    val result = new Array[Int](N)
    var i = 0
    while i < N do
      result(i) = ((i * 1103515245L + 12345L) & 0x7fffffffL).toInt
      i += 1
    result
  private val expectedSum =
    var result = 0L
    var i = 0
    while i < N do
      result += values(i)
      i += 1
    result
  private def benchmark(): Unit =
    val queue = new PriorityQueue[Int]()
    var repeat = 0
    while repeat < INNER_REPEAT do
      var i = 0
      while i < N do
        queue.enqueue(values(i))
        i += 1
      var total = 0L
      var last = Int.MaxValue
      while queue.nonEmpty do
        val current = queue.dequeue()
        if current > last then
          println(s"Error: heap order violated, current=$current last=$last")
          return
        total += current
        last = current
      if total != expectedSum then
        println(s"$total != $expectedSum")
      repeat += 1
  @main def runHeapSort(): Unit =
    var t = 1
    while t <= T do
      val startTime = System.nanoTime()
      var r = 0
      while r < OUTER_REPEAT do
        benchmark()
        r += 1
      val duration = System.nanoTime() - startTime
      println(s"round $t: ${duration / 1000}us")
      t += 1
```

Figure A.10: `HeapSort` benchmark

```scala
object SlidingWindow:
  private final val T = 100
  private final val OUTER_REPEAT = 100
  private final val INNER_REPEAT = 100
  private final val N = 4096
  private final val WINDOW_SIZE = 128
  private val values =
    val result = new Array[Int](N)
    var i = 0
    while i < N do
      result(i) = ((i * 1103515245L + 12345L) & 0x7fffffffL).toInt
      i += 1
    result
  private val expectedSum =
    var result = 0L
    var windowStart = 0
    while windowStart + WINDOW_SIZE <= N do
      var maximum = Int.MinValue
      var i = windowStart
      while i < windowStart + WINDOW_SIZE do
        if values(i) > maximum then maximum = values(i)
        i += 1
      result += maximum
      windowStart += 1
    result
  private def benchmark(): Unit =
    var repeat = 0
    val queue = new MaxQueue[Int](WINDOW_SIZE + 1)
    while repeat < INNER_REPEAT do
      queue.clear()
      var total = 0L
      var i = 0
      while i < N do
        queue.enqueue(values(i))
        if i >= WINDOW_SIZE then
          queue.dequeue(values(i - WINDOW_SIZE))
        if i >= WINDOW_SIZE - 1 then
          total += queue.getMax
        i += 1
      if total != expectedSum then
        println(s"Error: total=$total, expectedSum=$expectedSum")
      repeat += 1
  @main def runSlidingWindow(): Unit =
    var t = 1
    while t <= T do
      val startTime = System.nanoTime()
      var r = 0
      while r < OUTER_REPEAT do
        benchmark()
        r += 1
      val duration = System.nanoTime() - startTime
      println(s"round $t: ${duration / 1000}us")
      t += 1
```

Figure A.11: `SlidingWindow` benchmark