Master's project

Programming Methods Laboratory (LAMP)

Inline Traits for Specialization in Scala 3

Timothée Andres

supervised by

Dr. Nicolas Stucki

Pr. Martin Odersky

Spring 2023

Contents

1 Introduction

Many strongly typed programming languages have introduced polymorphism into their specifications, like Java [2], Scala [7] and C++ [1], to cite a few. This elegant construct makes code more factorizable and reusable when the type of the data is less important than how it is handled.

This very useful concept comes however with a drawback: genericity is a concept that does not exist at the machine's level. We therefore need a way for it to understand what a generic type is. Two solutions are widely used:

- the first, called *erasure*, is to represent all values with a single, homogenized representation, and replace generic types with it at compile time (e.g. `Object` in Java),
- the second, used for example in C++, is to use the generic code for homogeneous representations (reference types) and generate specialized code at runtime for heterogeneous values (value types).

1.1 Boxing and unboxing

In Java, primitive types cannot be used as arguments to a generic function: they do not conform to the "homogenized representation" mentioned earlier; instead, they have to be wrapped in an object. The process of wrapping a primitive type in its corresponding wrapper object is called *boxing*, while extracting a primitive type from its wrapper object is called *unboxing*. This process is most often done automatically by the compiler in a process called *autoboxing*: the user does not need to explicitly make the conversion between primitive type and object, shortening the code as a result.

This boxing mechanism, however, has a cost: operations that must be performed on primitive types, such as arithmetic operations (`+`, `-`, `*`, `/`, etc.), are delayed by the need to unbox the argument and rebox the result. For example, in Scala 3, if we compile the code from listing 1, erasure will generate the code presented in listing 2.

```scala
class Wrapper[T](val x: T)

class C:
  val w1 = Wrapper[Int](1)
  val w2 = Wrapper[Int](2)
  val w3 = Wrapper[Int](w1.x + w2.x)
```

Listing 1: Example of autoboxing in Scala 3

Despite its usefulness, boxing values come with a cost in performance: a simple operation on a primitive value turns into several method calls and object instantiations, as shown in listing 2. You can find a benchmark showcasing this effect in appendix A.

```
class Wrapper(x: Object) extends Object() {
  def x(): Object
}

class C() extends Object() {
  def w1(): Wrapper = new Wrapper(Int.box(1))
  def w2(): Wrapper = new Wrapper(Int.box(2))
  def w3(): Wrapper = new
       Wrapper(Int.box(Int.unbox(this.w1().x()).+(Int.unbox(this.w2().x()))))
}
```

Listing 2: Code generated from listing 1 after type erasure

1.2 Code specialization

1.2.1 Specialization in Scala 2

The concept of *code specialization* is a good way to circumvent this issue. While it is merely one of the solutions to this problem, it has been implemented into several languages to improve the performance of generic code.

Specialization is a form of *monomorphization*, the opposite of polymorphization. The goal is to generate concrete code from generic code, in order to have it behave more optimally than in the generic case. This can be used to avoid unnecessary boxing and unboxing.

Several forms of specialization exist, all tailored to the language in which they exist. For example, C++ offers it through templates that can be explicitly specialized [5] (see listing 3).

```cpp
#include <type_traits>

// primary template
template<typename T> struct is_void : std::false_type {};
// explicit specialization for T = void
template<> struct is_void<void> : std::true_type {};

int main()
{
    static_assert(is_void<char>::value == false,
        "for any type T other than void, the class is derived from false_type");
    static_assert(is_void<void>::value == true,
        "but when T is void, the class is derived from true_type");
}
```

Listing 3: Specialization of templates in C++

Templates are a powerful tool for specialization:

- the specialized code can be arbitrarily different from the generic one, as long as the signatures are coherent,
- the type for which it specializes can be arbitrarily complex, and
- templates also accept term parameters, which behave exactly as macro parameters in C.

Scala 2 offers a way to specialize generic types with the `@specialized` annotation[1]. Its goal is to generate specialized definitions of methods and classes at compile time and replace references to them with their specialized version when available [3]. The compiler can also be instructed to specialize only on a subset of the possible types, by passing corresponding type parameters to the annotation. An example of this is given in appendix B.

Templates in C++ and specialized classes in Scala are almost opposite in their behaviour:

- the `@specialized` annotation of Scala tells the compiler to perform its operations behind the scenes; users cannot write their own specialization of a generic class like in C++,
- C++ templates duplicate their code and replace the generic types on demand when the template is applied to a concrete type (generated at call site), whereas specialized classes in Scala 2 are generated when the generic class is compiled (generated at definition),
- templates can specialize code on any type, whereas `@specialized` only works on primitive types: `Boolean`, `Byte`, `Char`, `Double`, `Float`, `Int`, `Long`, `Short` and `Unit`.

1.2.2 Issues with `@specialized`

However, this annotation opens the door to multiple issues [6, 3]. Consider the following code, written in Scala 2:

```scala
import scala.specialized

class A[@specialized T] {
  val x: T = ???
}
class B[@specialized U] extends A[U] {}
```

Listing 4: Problematic specialized code

and let the specialization of class `X` for type `T` be named $\texttt{X}_\texttt{T}$. The following issues arise:

1. If a specialized class extends another specialized class, we encounter the inheritance diamond problem: $\texttt{A}_\texttt{Int}$ extends `A`, and since `B` extends `A`, we would like $\texttt{B}_\texttt{Int}$ to extend both `B` and $\texttt{A}_\texttt{Int}$. This is however impossible, due to the absence of multiple class inheritance in Scala[2],
2. Without restrictions on the types on which to specialize, we get 10 versions of the class: specialized versions for each of the primitive types, plus the original definition.

   This behaviour is repeated for every specialized generic types, which means that the number of classes generated is exponential: with two specialized

[1] The `@specialized` annotation has however not been ported to Scala 3. The symbol exists, but using it does not provide any specialization.

[2] Note that the compiler for Scala 2.13.6 warns of this issue: `"warning: class A must be a trait. Specialized version of class B will inherit generic A[Int]"` *[sic]*.

types, specialized classes must be generated for every pair of types listed above, and so on.

In general, if a class has n specialized types, $O(9^n)$ specialized classes will be generated alongside the generic one,

3. Since specialized classes need to extend the generic class, the latter needs to define specialized accessors for each of their generic values (see listing 5).

```
class A[@specialized T] extends Object {
  def <init>(): A[T] = {
    A.super.<init>();
    ()
  };
  protected[this] val x: T = scala.Predef.???();
  def x(): T = A.this.x;
  def x$mcZ$sp(): Boolean = A.this.x().asInstanceOf[Boolean]();
  def x$mcB$sp(): Byte = A.this.x().asInstanceOf[Byte]();
  def x$mcC$sp(): Char = A.this.x().asInstanceOf[Char]();
  def x$mcD$sp(): Double = A.this.x().asInstanceOf[Double]();
  def x$mcF$sp(): Float = A.this.x().asInstanceOf[Float]();
  def x$mcI$sp(): Int = A.this.x().asInstanceOf[Int]();
  def x$mcJ$sp(): Long = A.this.x().asInstanceOf[Long]();
  def x$mcS$sp(): Short = A.this.x().asInstanceOf[Short]();
  def x$mcV$sp(): Unit = { A.this.x(); () };
  def specInstance$(): Boolean = false
};
```

Listing 5: Class `A` after specialization

In general, if a class with specialized types has m generic members and n specialized types, $O(9^n \cdot m)$ specialized accessors will be generated alongside the generic one.

4. Similarly, specialized classes override the generic accessors of their parent, meaning that for every specialized definition in a specialized class, another definition for the generic accessor is generated as well (see listing 6).

```
class A$mcI$sp extends A[Int] {
  def <init>(): A$mcI$sp = {
    A$mcI$sp.super.<init>();
    ()
  };
  protected[this] val x$mcI$sp: Int = scala.Predef.???();
  def x$mcI$sp(): Int = A$mcI$sp.this.x$mcI$sp;
  override def x(): Int = A$mcI$sp.this.x$mcI$sp();
  def specInstance$(): Boolean = true
};
```

Listing 6: Specialized class $\texttt{A}_{\texttt{Int}}$

In general, if a class with specialized types has m generic members, each specialized class will have $O(2 \cdot m)$ members.

5. Finally, as shown in listing 6, a new specialized field (`x$mcI$sp` for Int) is created for the specialized classes and used instead of the default field `x`.

This means that space is reserved for this field in the JVM, but it is never used, leading to some space being wasted.

In general, if a class with specialized types has f generic fields, each specialized class will have $O(2 \cdot f)$ fields, half of which are never used.

A direct consequence of these issues could be called "code explosion": from a small generic class, we end up with a greater amount of longer classes, with many definitions used only for inheritance and fields that are never used.

```java
public class A<T> {
  public final T x;
  public T x();
  public boolean x$mcZ$sp();
  public byte x$mcB$sp();
  public char x$mcC$sp();
  public double x$mcD$sp();
  public float x$mcF$sp();
  public int x$mcI$sp();
  public long x$mcJ$sp();
  public short x$mcS$sp();
  public void x$mcV$sp();
  public boolean specInstance$();
  public A();
}
```

Listing 7: Java code for class `A` from listing 4

```java
public class A<T> {
  private final T x;
  public T x();
  public A();
}
```

Listing 8: Java code for class `A` without `@specialized` annotation

1.2.3 Desired solutions

The diamond issue is caused by the Java rule that $\texttt{B}_{\texttt{Int}}$ cannot extend both classes `B` and $\texttt{A}_{\texttt{Int}}$. However, we know that the two classes are related to $\texttt{B}_{\texttt{Int}}$, and it would be desirable to express this.

We also see that if multiple specialized types are used in a single class, there might be more classes generated than strictly necessary, and space will be lost due to the duplication of fields in the specialized classes. We therefore need a way to mitigate this, and give the user the ability to explicitly specify for which combination of types to specialize.

Finally, in order to reduce the amount of code generated, it would be convenient for the code to be specialized on demand, similarly to templates in C++: if the specialized code is never used, we do not want to generate any additional code.

1.3 Inlining in Scala

In Scala 2, the `@inline` annotation can be used as a hint for the compiler that a method should be inlined. However, this remains a hint, not a directive: as stated in the Scala 2.13.4 API documentation, "by default, the Scala optimizer is disabled and no callsites are inlined." [8].

Scala 3 introduced the soft keyword `inline` as a replacement for the annotation. Instead of providing the compiler with hints, the user is given control over what get inlined: "[inlining] can be seen as a form of metaprogramming, where inlining is the syntactic construct that turns a program into a program generator" [10, p. 14].

The inliner performs the following task: when a call to a method is found, it checks if the definition needs to be inlined. If so, it adapts the right-hand side of the method, and inlines it at the call site.

Combined with macros, `inline` provides Scala users with some form of metaprogramming: the compiler can generate and/or run arbitrary code at compile time, which provides additional flexibility to programs.

The inlining phase already performs some form of specialization: since the call site of an inline method may be arbitrarily different and far from its definition, the environment might be completely different in both cases. To remedy this, the inliner retypes and adapts the code when it is inlined.

2 Inline traits

We introduce a new kind of structure that inlines its content into the classes and traits that extend it, called *inline traits*. If, at the time of inlining, we know more about the generic types, the compiler inlines the definitions and specializes the types.

Here is an example:

```scala
inline trait A[T](val x: T):
  def foo: T = x

class B extends A[Int](1)
```

Listing 9: Inline trait

After the *inlining* phase, the code becomes the following:

```scala
package <empty> {
  inline trait A[T >: Nothing <: Any](x: T) extends Object {
    T
    val x: T
    def foo: A.this.T = this.x
  }
  class B() extends Object(), A[Int](1) {
    override val x: Int = 1
    override def foo: Int = this.x
  }
}
```

Listing 10: Code of listing 9 after inlining

As we can see, the field `B.x` has been specialized to be an `Int`. This means that operations on this field will not require boxing and unboxing operations if they expect an `Int`. A more complete example showcasing the differences in code generated by the compiler with and without the `inline` keyword is available in appendix C.

If we were to combine this new construct with the macros of Scala 3, we can create a class whose content may be completely different depending on the type arguments passed to its parents. For example, listing 11 shows what could be an SQL query builder that generates code depending on the content of the query.

2.1 Leveraging the inlining engine

As explained before, the inlining mechanism of Scala 3 already performs some form of specialization. This is illustrated in listing 12: a method from object `A` is inlined inside of object `B`, and the generic types are known at this point. The inliner adapts the code so that the code works without autoboxing.

Since there is already code responsible for managing references to `this`, owners of declarations, types of inline methods, and more, it stands to reason that the first version of inline traits should leverage that power for its purpose.

```scala
// Macro that generates code depending on the content of an SQL query
inline def optimizedQuery[Q <: String & Singleton]: Query =
  ${ compileQueryExpr[Q] }
def compileQueryExpr[Q <: String & Singleton](using Quotes): Expr[Query] =
  queryExprOf(optimizeSQL(analyzeSQL(parseSQL(Type.valueOfConstant[Q]))))

inline trait SqlQuery[Q <: String & Singleton]:
  private val optimizedQuery: Query = compileQuery[Q]
  def run(): Unit = new DBConnection().run(query)

// The code generated for these two classes may be arbitrarily different
final class SqlGetUsers extends SqlQuery["SELECT * FROM users;"]
final class SqlDropUselessTable extends SqlQuery["DROP TABLE uselessTable;"]
```

Listing 11: Pseudocode of specialized query builder template

```scala
class Wrapper[T](val x: T)
class IntWrapper(override val x: Int) extends Wrapper[Int](x)

inline def foo[T](w: Wrapper[T]): T = w.x

val wrap = IntWrapper(1)

val i: Int = foo(wrap)
// This will be replaced with code equivalent to:
//   val i: Int =
//     val w = wrap
//     w.x
// Here, the compiler will not unbox the result: even though foo is generic, we
    know that wrap.x is an Int and not a T
```

Listing 12: Specialization of inlined code

2.2 Design choices

Let us define a *class-like* to be something that can be represented by a `TypeDef` with a `Template` in its right-hand side; in other words, a class-like is one of the following: trait, class, object, or enum.

2.2.1 Using traits to avoid `@specialized` issues

As described in section 1.2.3, we wish to fix the issues posed by `@specialized` in Scala 2, namely the inheritance diamond problem, code explosion, and unused fields.

The diamond problem is automatically fixed by enforcing the use of traits: class-likes can extend any number of traits.

+ class-likes can both be specialized and extend all of their inline trait ancestors

− restrictions on traits also apply to inline traits

Regarding code explosion, we choose to never inline code inside of an inline trait. This way, if an inline trait `B` extends another inline trait `A`, no code will be inlined inside of `B`; if class `C` extends B however, the code of both `A` and `B` will be inlined and specialized inside that class.

+ if an inline trait is never extended, no additional code is generated
+ code is always inlined in non-inline children, allowing for complex inheritance trees between inline traits without code explosion
− code is duplicated for each child of inline traits

Finally, the inlining of an inline trait's members inside its children means that no fields will be duplicated: the overriding mechanism will ensure that only one field is generated in the bytecode.

+ no unused fields in specialized children

2.2.2 Inlining the body of an inline trait

Public and protected members Member definitions inside of an inline trait are to be inlined in its first non-inline descendant. These definitions may contain references to generic types or to `this`, which need to be adapted to their new definition site:

- a generic type that is known in the new environment is replaced with its concrete type
- a reference to the inline trait instance through the `this` keyword is replaced with a reference to the extending class-like

An example is given in listing 13.

```
inline trait A[T]:
  val x: Int = 1
  val t: T
  def foo(): Int = this.x
  def bar(): T = this.t

class B extends A[Boolean]:
  override val t: Boolean = true
  // The following members are generated by the compiler
  override val x: Int = 1
  override def foo(): Int = this.x
  override def bar(): Boolean = this.t
```

Listing 13: Adapting inlined code

Private members An inline trait can define, in its body, private members just like a normal trait would. However, this means that they need to be treated slightly differently than members accessible from outside of the instance.

They cannot be marked as overridden, as they are not visible from the extending class, but they still need to be present for the inlined code to work as expected. Furthermore, if an inline trait declares a private field, it should not interfere with other fields of the extending class-like or of another parent.

To ensure this, we inline private members without the `override` keyword, and we rename them with a name that is unique in the context of the child class.

An example is given in listing 14.

```
inline trait A(b: Boolean):
  private val x: Int = 1
  def foo(): Int = if b then x + 1 else 0

class B extends A(true):
  // The following members are generated by the compiler
  private val A$$b: Boolean = true
  private val A$$x: Int = 1
  override def foo(): Int = if this.A$$b then this.A$$x.+(1) else 0
```

Listing 14: Inlining of private fields

Pruning members after inlining Since definitions are never accessed directly on an inline trait, but on one of its implementations, the right-hand side of the definitions are not necessary: once we know that the right-hand sides are correct, and the definitions are stored in the corresponding TASTy file, we may simply prune them from the inline trait to reclaim some space.

An example is given in listing 15.

```
inline trait A:
  val x: Int = 1
  def foo(i: Int): Int = x + i

// After compilation, the two definitions will be
//   val x: Int
//   def foo(i: Int): Int
```

Listing 15: Pruning the right-hand sides

2.2.3 Body statements

In Scala, a trait's body may contain statements which are not member definitions. Those are placed by the compiler into the initializer `$init` of the trait (see listings 16 and 17).

```
trait A:
  def foo() = println("I am A!")
  foo()

class B extends A:
  def bar() = println("I am B!")
  bar()
```

Listing 16: Statements in a normal trait's body

We can reproduce this behavior by simply inlining statements of the inline trait before those of the extending class, as long as the inlining takes place before the *constructors* phase. Since we know that inline traits are not to be instantiated directly, we may remove their bodies' statements once their code is inlined (see listings 18 and 19).

```
trait A extends Object {
  def $init(): Unit =
    {
      this.foo()
      ()
    }
  def foo(): Unit = println("I am A!")
}

class B extends Object, A {
  def <init>(): Unit =
    {
      super()
      super[A].$init()
      this.bar()
      ()
    }
  def foo(): Unit = super[A].foo()
  def bar(): Unit = println("I am B!")
}
```

Listing 17: Code of listing 16 after *Constructors* phase

```
inline trait A:
  def foo() = println("I am A!") // The right-hand side will be pruned
  foo() // This statement will be removed

class B extends A:
  def bar() = println("I am B!")
  bar()
```

Listing 18: Statements in an inline trait's body

```
inline trait A extends Object {
  def foo(): Unit
}
class B extends Object, A {
  def <init>(): Unit =
    {
      super()
      this.foo()
      this.bar()
      ()
    }
  override def foo(): Unit = println("I am A!")
  def bar(): Unit = println("I am B!")
}
```

Listing 19: Code of listing 18 after *Constructors* phase

2.2.4 Inner classes

Since we wish to inline the code from the inline trait into the children class-likes, we need to take care that specialized inner class-likes are compatible with the original ones. However, in Scala, class definitions cannot be overridden[3]. This poses an issue, as we would like for an inlined inner class-like to:

- have the same name as the original inner class-like, and
- extend the original inner class-like to preserve typing relations.

In the following paragraphs, the different examples refer to the code of listing 20, and the final result can be found in listing 21.

```scala
class Model[T](val m: T)

inline trait A[T](x: T):
  class Inner[U](t: T, u: U) extends Model[T](t):
    def this(u: U) =
      this(x, u)
      println("Without t")
    def foo(): (T, U) = (t, u)

class B extends A[Boolean](true)
```

Listing 20: Example of inner class

```scala
class Model[T >: Nothing <: Any](m: T) extends Object() {
  T
  val m: T
}

inline trait A[T >: Nothing <: Any](x: T) extends Object {
  T
  private[this] val x: T
  trait Inner$trait[U](t: T, u: U) extends Model[T]:
    U
    def foo(): Tuple2[A.this.T, Inner$trait.this.U] =
      Tuple2.apply(this.t, this.u)
  type Inner[U] <: Inner$trait[U]
  def new$Inner[U](t: T, u: U): Inner[U]
  def new$Inner[U](u: U): Inner[U]
}

class B() extends Object(), A[Boolean](true) {
  private[this] val A$$x: Boolean = true
  class Inner[U](t: Boolean, u: U) extends Inner$trait[U], Model[Boolean](t):
    U
    def this(u: U) =
      this(B.this.A$$x, u)
      println("Without t")
    def foo(): Tuple2[Boolean, Inner.this.U] = Tuple2.apply(this.t, this.u)
  def new$Inner[U](t: T, u: U): Inner[U] = new Inner[U](t, u)
  def new$Inner[U](u: U): Inner[U] = new Inner[U](u)
}
```

Listing 21: Code of listing 20 after inlining A

[3]In Scala 2, class definitions could be shadowed, but this has been deprecated in Scala 3 [4].

Inner trait Inner class-likes are turned into inner traits and their name is changed to avoid conflict with their inlined counterparts (for example, `class Inner` is transformed into `trait Inner$trait`).

The declaration of the parents of the inner class-like are adapted to fit the signature of a trait: if the inner class-like has as a parent a trait, it cannot call the trait's constructor anymore [9], and term parameters must be pruned and left for the inlined inner class-like to pass. Furthermore, secondary constructors are pruned, for they are forbidden in traits.

In listing 20, if we try to generate a class `Inner[U]` inside of `B`, we would need it to extend both `Model[U]` (for consistency with the original class) and `A#Inner[U]` (to preserve typing relations). This is another instance of the inheritance diamond problem, which we solve in a similar way as previously: we transform the inner class-like into a trait with a similar signature.

```
inline trait A[T >: Nothing <: Any](x: T) extends Object {
  // [...]
  trait Inner$trait[U](t: T, u: U) extends Model[T]
  // [...]
}
```

Listing 22: Transformation of `class Inner` into `trait Inner$trait`

This choice preserves the code of the inner class-like while solving the diamond problem for the inlined definition. However, secondary constructors cannot be preserved in this form [9]; we explain below how to handle them.

Furthermore, if the signature of the class-like cannot be preserved during the transformation due to restrictions on traits[4], this technique will not work. In the first version of inline traits, we forbid such inner class-likes.

Objects need another manipulation to be transformed. In Scala 3, an object is desugared to a class that can be instantiated only once by its companion `lazy val`:

```
object Obj

// The compiler desugars the object into
//   lazy val Obj = new Obj$
//   class Obj$:
//     self: Obj.type =>
```

Listing 23: Desugaring of an object by the compiler

During the phase responsible for specializing the inline traits, the lazy val needs to be marked as non-lazy and both the val and the class need to lose their `module` flag: since we are transforming the object into a trait, they should not be marked as if they were the result of desugaring an object anymore.

Their inlined counterpart will, however, possess the `module` flag and the val will be lazy.

+ inlined inner class-likes can inherit both from the original class-like as well as its original parents

[4]For example, a trait's constructor cannot accept by-name parameters.

– inner class-likes whose signatures are incompatible with the restrictions on trait signatures cannot be transformed this way

Type A new type is created with the inner class-like's name, with as upper bound the inner trait previously transformed:

```
inline trait A[T >: Nothing <: Any](x: T) extends Object {
  // [...]
  type Inner[U] <: Inner$trait[U]
  // [...]
}
```

Listing 24: Generation of type `Inner`

This type serves as a way for the typing hierarchy to know that the inlined inner class-like is a subtype of the original one: when the compiler sees in `B` a class-like called `Inner`, it assumes that it is a concrete implementation of this type, which acts as a proxy for the subtyping relationship between the inner class-likes.

+ inlined inner class-likes can inherit from their original inner class-likes

– the size of the inline trait's code is slightly increased

– we need to be careful about the instantiation of inner class-likes, as the name now references a type (see below)

Constructors proxies If the inner class-like has constructors, they cannot be kept in the resulting inner trait: in Scala 3, traits cannot have secondary constructors, and trait constructors may not be called to create new instances [9]. We therefore need to create proxies for the code to stay correct.

For each constructor of the inner class-like, we generate a new method that acts as its proxy, with a name related to the inner class-like's name and a signature resembling as closely as possible the constructor's signature.

For example, with the code of listing 20, we would generate the following methods in inline trait `A`:

```
inline trait A[T >: Nothing <: Any](x: T) extends Object {
  // [...]
  def new$Inner[U](t: T, u: U): Inner[U]
  def new$Inner[U](u: U): Inner[U]
  // [...]
}
```

Listing 25: Proxy methods for constructors

Note that the constructors proxies are left abstract. The inner class-like having been turned into a trait, we can not instantiate it anymore. The task of implementing the proxies is delegated to the concrete children of the inline trait.

+ secondary constructors are preserved

– references to the inner class-like's constructors need to be adapted everywhere to point to the proxy methods

– the size of the inline trait's code is increased proportionally to the number of constructors the inner class-like has

Inlined code Finally, all that remains is to inline the inner class-like inside of B, and implement the constructor proxies. Note that the new class-like extends both the original class-like as well as its parents to satisfy the bound constraint of the type `Inner` shown in listing 24:

```scala
class B() extends Object(), A[Boolean](true) {
  private[this] val A$$x: Boolean = true
  class Inner[U](t: Boolean, u: U) extends Inner$trait[U], Model[Boolean](t):
    U
    def this(u: U) =
      this(B.this.A$$x, u)
      println("Without t")
    def foo(): Tuple2[Boolean, Inner.this.U] = Tuple2.apply(this.t, this.u)
  def new$Inner[U](t: T, u: U): Inner[U] = new Inner[U](t, u)
  def new$Inner[U](u: U): Inner[U] = new Inner[U](u)
}
```

Listing 26: Inlining of `A` inside of `B`

2.3 Accessing members through an inline trait

This specification chooses to have inline traits behave as closely as possible to traits. However, this raises the following question: should we allow accessing methods and values on inline traits?

```scala
inline trait A[+T](val x: T)
class B(i: Int) extends A[Int](i)

val as: Seq[A[Int]] = (1 to 5).map(i => B(i))
val sum = as.foldLeft(0){
  case (s: Int, a: A[Int]) => s + a.x
}
```

Listing 27: Accessing `x` through `A`

We propose the following solutions:

1. forbid the code of listing 27, and ask the user to change the type of `as` to `Seq[X]`, where `X` is a non-inline subclass of `A`,
2. let virtual dispatch redirect the call to `B.x` (specialization is lost, as we access `x` through the generic environment of `A`),
3. narrow the type of `as` to `Seq[B]`,
4. generate specialized, non-inline children of `A`, and replace the types accordingly (akin to what `@specialized` does in Scala 2).

3 Implementation

The following implementation of inline traits can be found on the official dotty GitHub repo, in pull request #17329.

3.1 Allowing member access through inline traits

In regards to the design problem presented in section 2.3, we chose to implement solution 2: we will allow users to write code similar to the one in listing 27, where members are accessed through an inline trait type, at the cost of losing the specialization.

3.2 Leveraging existing code for inlining

As previously mentioned in section 2.1, the compiler is already able to properly inline code from one scope to another [10]. The inlining of method calls is currently done in two different phases:

- in the *typer* phase, for `transparent inline` members,
- in the *inlining* phase, for other `inline` members.

The code is however shared between these phases, and split between three source files:

- **Inlining.scala**, containing the class `Inlining` representing the compiler phase,
- **Inliner.scala**, containing the class `Inliner` capable of inlining a method call, and
- **Inlines.scala**, containing an object `Inlines` defining helper methods and classes, among which `InlineCall`, a descendant of `Inliner`.

The code to be inlined may be inaccessible during the *inlining* phase of the compilation, so the compiler adds a `@BodyAnnotation` annotation that contains the method's right-hand side to the inline method's symbol. This way, the inliner simply has to verify if a method call is done on a method that has such an annotation, and if so, retrieves the body stored inside and passes further for inlining.

When inlining a method call, two elements are capital:

- the call itself, containing information about the call site context, and
- the body of the inline method, to be adapted and inlined.

We may reuse the code responsible for this by "cheating" the inlining engine:

- we place the body of the inline trait inside of a `@BodyAnnotation` in order to make it retrievable, by wrapping its statements inside of a `Block` and using a unit literal as the last statement,
- we pass the parent definition in lieu of the method call: for example, if the extending class-like is defined as `class B extends A[Int]`, we pass `A[Int]` as if it were the method call,

- we retrieve the inline trait's body from its annotation, and for each definition that needs to be inlined, we adapt it to the descendant class,
- we add the resulting specialized code to the child class-like's body.

This is done in **Inlines.scala**, in the method `inlineParentInlineTraits` and the class `InlineParentTrait`.

Instead of adding the new inlining code inside of the existing *inlining* phase, we created two new phases *specializeInlineTraits* and *pruneInlineTraits* which will implement the specification described in section 2.2.

3.3 File Inlines.scala

3.3.1 Method `Inlines.inlineParentInlineTraits`

This method takes as argument the tree of a class-like extending an inline trait. It does the following:

1. compute the symbols overridden by the class-like's definitions; if the user overrides methods and fields, we do not want to inline the original definitions,
2. for each inline trait ancestor that has not been inlined in a parent yet[5]:
   (a) compute the symbols that have already been overridden, both by the extending class-like and by previous inline trait ancestors,
   (b) adapt the code to the extending class-like's environment,
   (c) update the references to symbols that were inlined, such as inner class-likes,
3. recreate the body of the child class-like to contain the inlined code as well as the updated original code.

The order in which ancestors are inlined is important: in Scala 3, inheritance precedence is computed with a left-leaning DFS algorithm, as illustrated in listing 28.

The current inlining algorithm trusts that `cls.tpe.baseClasses` returns the ancestors in descending order of precedence (in listing 28: `C`, `P2`, `GP3`, etc.), where `cls` is the tree of a class-like extending some inline traits.

3.3.2 Method `InlineParentTrait.expandDefs`

This method is responsible for the creation of the code to be inlined inside of the child class-like. It takes as argument the list of symbols which are already overridden, and returns the list of `tpd.Tree` to be inserted in the class-like's template.

It performs the following steps:

[5]If a class-like extends an inline trait that itself extends another inline trait, the grandparent has not been inlined in the parent and therefore needs to be specialized in the child.

```scala
trait GP1
trait GP2
trait GP3

trait P1 extends GP1, GP2
trait P2 extends GP2, GP3

class C extends P1, P2

//  GP1  GP2  GP3
//    \  / \  /
//     P1   P2
//      \   /
//       \ /
//        C
//
// Inheritance precedence: GP1 < GP2 < P1 < GP3 < P2 < C
```

Listing 28: Order of inheritance in Scala 3

1. Register the term arguments passed to the inline trait's constructor in the helper class `ParamAccessorsMapper`. These will be used in `inlinedValDef` (section 3.3.4), when we generate the member definitions for parameter accessors. For example, when compiling the following code, we register `3` in the mapper as the value of `x`:

```scala
inline trait A[T](x: T)

class B extends A[Int](3)
```

2. Retrieve the parent's body stored in its `@BodyAnnotation`, and filter out the definition previously overridden.

3. Generate new symbols for each member definition to inline; this is due to the way the inlining is done: when we inline the right-hand side of a definition, we might create a reference to a private field that was not inlined yet, and has no symbol in the child class.

   For example, in the code of listing 29, if we were to inline the definitions before creating the new symbols, we would create a new symbol for `x` and enter it in `B`, then adapt its right-hand side to be `B.this.y`. However, at this point of the compilation, `B` does not contain a symbol `y`, and the definition of `y` in `A` is private, resulting in an error raised by the typer.

4. Adapt each statement of the parent's body:

   - if the statement is a member definition (`val`, `var`, `def`, `class`, `type`, etc.), call `expandStat` to inline its right-hand side,
   - otherwise, inline the statement directly.

3.3.3 Symbol-inlining methods

`inlinedClassSym` Creates a new symbol for an inner class-like that will be inlined.

```scala
inline trait A:
  val x: Int = A.this.y
  val y: Int = A.this.x

class B extends A
```

Listing 29: Circular references in inline trait

As stated in section 2.2.4, we wish for the inlined inner class-like to derive from the original one, so we need to make sure to create a new parent to represent this relationship.

We also need to create a new scope for this symbol: we will enter the new member symbols afterwards, therefore we must not preserve the old symbols in it.

Finally, we register in the map `innerClassNewSyms` that the inlined inner class-like is a specialized version of the original one, we enter the symbol into the extending class-like, and we return the symbol.

`inlinedMemberSym` Creates a new symbol for definitions that are not inner class-likes.

Most of the work done here is computing the new symbol's flags:

- if the member is non-private, we add the `override` flag to the new symbol,
- if the member is a parameter accessor, we remove the `ParamAccessor` flag,
- if it is a local parameter accessor (i.e. it does not have a `val` or `var` keyword in its definition), we generate a new name for it based on the parent's name (for example, a local parameter `x` from an inline trait `A` is renamed to `A$$x`) and register it so that references to it may be updated to point to the new symbol,

Once this is done, we adapt the symbol's info to the context of the child class-like and return it. An example of adapting the symbol's info is to replace references to generic types by the concrete type of the child's class:

```scala
inline trait A[T]:
  val foo: T = ???

class B extends A[Int]
// inlinedMemberSym will make the inlined symbol of foo a value of type Int
```

3.3.4 Definition-inlining methods

In the following section, when "inlining the body" of a definition is mentioned, it means passing the definition's right-hand side to the `inlinedRhs` methods:

- `inlinedRhs(ValOrDefDef, Symbol)`, used to inline the right-hand side of a `val` or `def` definition that already has an inlined symbol, and
- `inlinedRhs(Tree)`, used to inline a tree by relying on the code of the `Inliner` class.

The first method changes the owner of the right-hand side to the new symbol, and calls the second method.

`inlinedValDef` Inlines a `val` definition.

If it is a parameter accessor, we use its value provided in the class-like's signature as a right-hand side, otherwise we inline its body. An example is given in listing 30.

```
inline trait A[T, U](x: T, var v: U):
  val y: Int =
    val i = 1
    2 * i
  var z: Boolean = this.y > 0

class B extends A[Int, Double](3, -1.2d):
// inlinedValDef will generate the following definitions:
  // The 3 and -1.2d here are taken from ‘extends A[Int, Double](3, -1.2d)‘
  private[this] val A$$x: Int = 3
  override var v: Double = -1.2d
  override val y: Int =
    {
      val i: Int = 1
      2.*(i)
    }
  override var z: Boolean = this.y.>(0)
```

Listing 30: Example of value inlining

Note that the `var` definitions are overridden. The rules of the compiler have been relaxed to allow this, since we assume that this field will never be accessed on the inline trait, but always on concrete implementations.

`inlinedDefDef` Inlines a `def` definition. An example is given in listing 31.

If the definition is a setter, we simply use `()` as its right-hand side, otherwise we inline its body.

```
inline trait A[T](var x: T):
  var y: Int = 0
  def foo(b: Boolean): T = A.this.x

class B extends A[Int](3):
// inlinedDefDef will generate the following definitions:
  override def x_=(x$1: Int): Unit = ()
  override def y_=(x$1: Int): Unit = ()
  override def foo(b: Boolean): Int = B.this.x
```

Listing 31: Example of definition inlining

This is due to the fact that during the *typer* phase, the compiler adds definitions for the setters of variables, with a unit literal as a right-hand side:

```scala
var x: Int = 1
// The typer generates the following definition:
def x_=(x$1: Int): Unit = ()
```

Listing 32: Generation of a setter definition

`inlinedTypeDef` Inlines a `type` definition. An example is given in listing 33.

This is one of the simpler helper methods, as it only needs to create a new `TypeDef` and assign to it the previously created symbol. Note that this does not include types defined in the inline trait's signature: those definitions are not inlined, because the types are directly replaced in the symbol's infos.

```scala
inline trait A[T, U, V <: U]:
  type C = Int
  type D >: T
  type E <: U
  type F >: V <: U

class B extends A[Boolean, AnyVal, Char]:
// inlinedTypeDef will generate the following definitions:
  type C = Int
  type D >: Boolean
  type E <: AnyVal
  type F >: Char <: AnyVal
```

Listing 33: Example of type inlining

`inlinedClassDef` Inlines a class-like definition.

It inlines the class-like in two steps, specializing the primary constructor first, then the class-like's body statements. Finally, the newly created definitions are used in a new `ClassDef`, which is directly passed to the code of `Inliner` to be adapted.

Note: This method is still a work in progress; only inner traits work for now.

3.4 File SpecializeInlineTraits.scala

The new phase *specializeInlineTraits* is run between the *pickler* phase and the *inlining* phase:

- after the *pickler* phase, because we wish to generate new code after the original code has been pickled. There is no need for the pickler to know about the internal changes made to inline traits and their extending classes,
- before the *inlining* phase, because we wish for the inline method calls to be inlined inside of the extending classes, not in the inline trait.

Its purpose is to transform:

- inline traits, so that their inner class-likes are rewritten according the the specification given in section 2.2.4, and
- class-likes that extend inline traits, so that the body of their parents are inlined in them.

3.5 File PruneInlineTraits.scala

The new phase *pruneInlineTraits* is run alongside the *pruneErasedDefs* miniphase. Its purpose is simply to remove the right-hand sides of the definitions of inline traits. We run this phase at this point of the compilation because of the following reasons:

- we wish to let as many checks be run on the inline trait's body before we erase the right-hand sides, and
- we decided to place the two pruning phases side by side.

Since we assume that the members of an inline trait can not be accessed directly, i.e. all members of an inline trait will always have been inlined in its children, we only need to keep the signatures in order for the JVM to know about them. This also reduces the size of the final code for the inline trait.

4 Evaluation

4.1 Code performance

When using inline traits to avoid the boxing/unboxing problem, we wish for it to be as performant as the `@specialized` annotation was in Scala 2. In the following subsections, we show the results of two benchmarks, run with the following parameters:

- 3 warm-up iterations of 5 seconds
- 5 measurement iterations of 10 seconds
- benchmark performed 10 times, using 3 threads

For each benchmark, three versions of the code have been used:

- one using the current specification of Scala 3, called *standard*; no specialization is done,
- one emulating the behaviour of `@specialized` in Scala 2, called *specialized* (more on this below), and
- one making use of inline traits, called *inlinetrait*.

Due to the fact that `@specialized` has been disabled in Scala 3, the code used in the *specialized* implementations has been manually specialized. The Scala 2.13.4 compiler was used to compile the code with the `@specialized` annotation, which served as inspiration for the manually specialized Scala 3 version of the code.

4.1.1 Matrix library

One of the ideas presented in the original discussion about inline traits is a matrix library, which would specialize its code based on the elements' type [6]. The three implementations can be found in appendix D.1.

Two random matrices of size 100×100 are generated, `m1` and `m2`, and the benchmark performs the following operation as many times as it can during the allotted iteration duration: `(m1 + m2) * m1`. The algorithms used for the sum and the product of two matrices are naive: assuming that the two operands are square matrices of size $n \times n$, the addition runs in $O(n^2)$ operations and the multiplication in $O(n^3)$ operations.

The results of the benchmark are presented in figure 1.

4.1.2 Specialized pairs

This benchmark was written in order to show that inline traits can be as efficient as the `@specialized` annotation's behavior, while resulting in less code generated. The three implementations can be found in appendix D.2.

We wish to perform operations on two types of pairs:

- pairs of the form `(i: Int, d: Double)`
- pairs of the form `(c: Char, s: Short)`

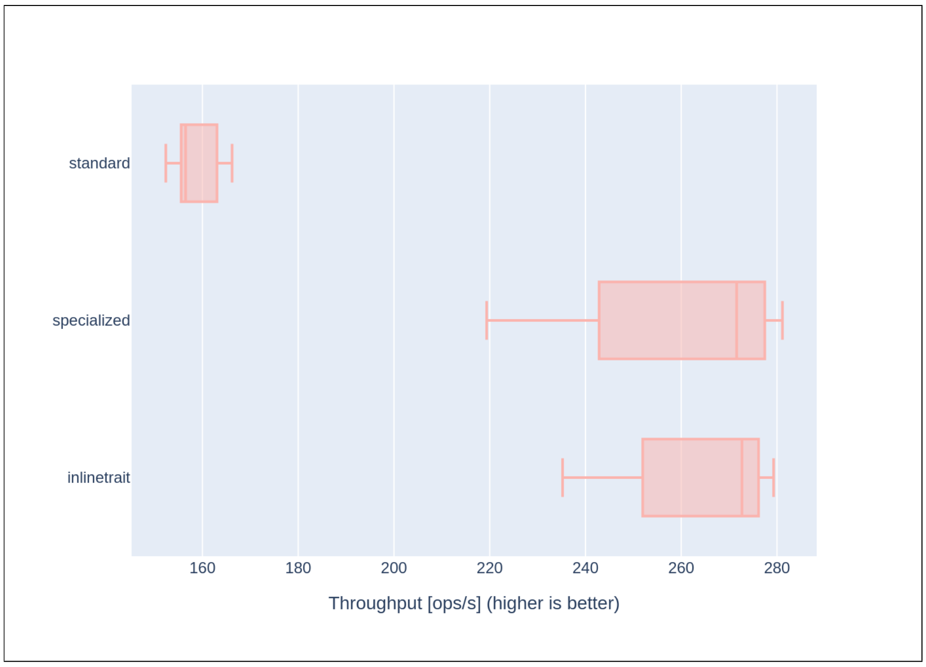

Figure 1: Number of execution per second on matrices operation

Three million random such pairs are generated, and passed through code equivalent to this one:

```scala
pairs.foldLeft(0){ case (sum, pair) => pair match {
  case Pair(i: Int, d: Double) => 7 * i + 3 * d.toInt + sum
  case Pair(c: Char, s: Short) => 5 * c + 2 * s + sum
}
```

The results of the benchmark are presented in figure 2. Note that the performances of the *specialized* and *inlinetrait* implementations are comparable, but the resulting code was shorter in the second case: the *specialized* code generated five pair classes:

- generic `Pair`
- specialized `Pair[Int, Double]`
- specialized `Pair[Int, Short]`
- specialized `Pair[Char, Double]`
- specialized `Pair[Char, Short]`

whereas the *inlinetrait* code only generated three:

- generic `Pair`
- specialized `Pair[Int, Double]`
- specialized `Pair[Char, Short]`

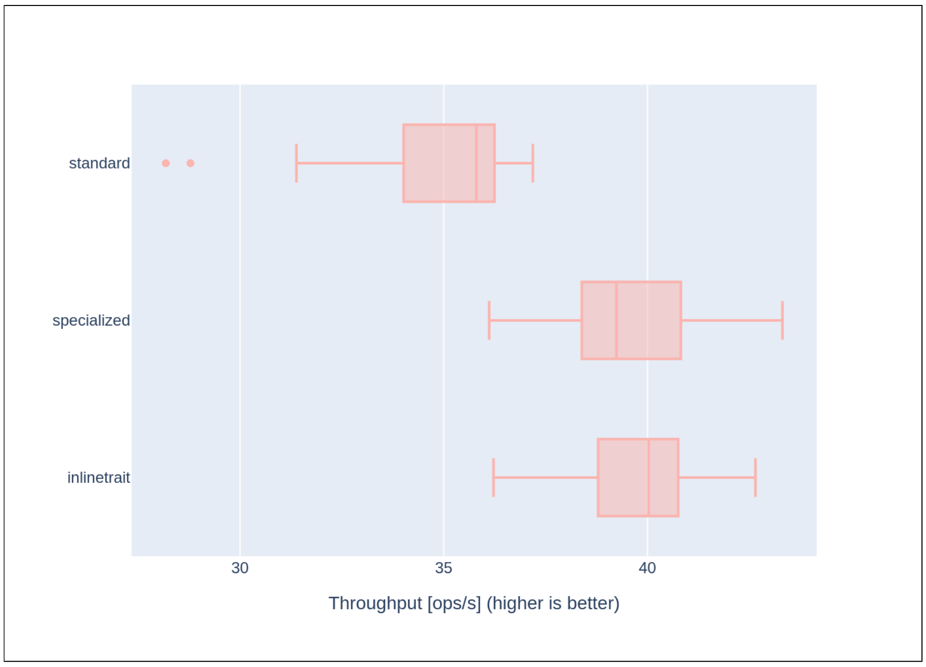

Figure 2: Number of execution per second on pairs operation

Conclusion

As the benchmark results show, the performance of inline traits is comparable to how `@specialized` used to perform in Scala 2, both of which are faster than making use of no specialization at all. We conclude that inline traits are a good replacement for `@specialized` in Scala 3 when the performance of generic code must be optimized.

4.2 Code size

One of the more important issues with `@specialized` was the amount of code generated. In order to see when inline traits has a different space efficiency, we compile several files containing similar code in Scala 2 and Scala 3, the former using `@specialize` and the latter inline traits.

4.2.1 Number of specialized classes

We create a base class/inline trait, called `A`, with three generic types over which to specialize: `T`, `U` and `V`. In each version of the code, we will add more specialized instances in order to see where the inline trait fares better than its annotation counterpart.

The metrics chosen to measure the amount of code generated are the number of class files generated, the size of the generic class file in bytes, and the total size of the class files in bytes.

We generate code with the narrowest specialization possible over the following types:

1. no specialization
2. `T = Int`
3. `T = Int`, `U = Int`, `V = Int`
4. `T = Int`, `U = Double`, `V = Boolean`
5. `T = Int`, `U = (Double or Int)`, `V = Boolean`
6. (`T = Int`, `U = Int`, `V = Int`) or (`T = Double`, `U = Double`, `V = Double`)
7. all possible types

The code for each case is provided in appendix D.3.

Types	Version	# classes	Size main class		Total size	
1	`@specialized`	1	1.6KiB		1.6KiB	
	`inline trait`	1	582B	(35%)	582B	(35%)
2	`@specialized`	2	1.9KiB		2.8KiB	
	`inline trait`	2	582B	(30%)	2.1KiB	(75%)
3	`@specialized`	2	2.1KiB		3.5KiB	
	`inline trait`	2	582B	(27%)	2.2KiB	(63%)
4	`@specialized`	2	2.3KiB		3.8KiB	
	`inline trait`	2	582B	(25%)	2.3KiB	(61%)
5	`@specialized`	3	2.4KiB		5.3KiB	
	`inline trait`	3	582B	(24%)	4.0KiB	(75%)
6	`@specialized`	9	2.5KiB		14KiB	
	`inline trait`	3	582B	(23%)	3.8KiB	(27%)
7	`@specialized`	730	4.3KiB		1.1MiB	
	`inline trait`	730	582B	(13%)	1.3MiB	(118%)

Table 1: Comparison of class sizes when specializing on various types

As table 1 shows, the size of the class file of `A` does not change when `A` is an inline trait. This is consistent with the specification, as no code is added to the body of an inline trait when classes extend it. To the contrary, we see that the size of the class `A` that uses the `@specialized` annotation grows with the number of classes that it needs to generate.

Another interesting figure is the number of class files generated: they are all identical, except for one specific case: when the tuples of specialized types are completely distinct. This results in the largest difference in total size, for a reduction in total size of almost one-fourth.

The only case where inline traits generate longer code than `@specialized` is when we specialize on all possible primitive types. This is however a case that should never arise, as this requires the user to write $9^3 = 729$ class definitions for 3 generic types, which seems unrealistic. On the off chance that specialization is required on all types *and* the amount of generated code must be minimal, we recommend to either use Scala 2 and the `@specialized` annotation, or to generate code using the macros of Scala 3.

4.2.2 Number of duplicated specialization

In terms of code generation, inline traits have one downside compared to the `@specialized` annotation: since code is inlined every time the inline trait is extended, it results in code duplication across all children of inline traits.

We create two source files, each containing one of the following definitions:

```scala
import scala.specialized
class A[@specialized T] {
  val x: T = ???
  val y: T = ???
  val z = (x, y)

  def foo(i: Int, j: Double): T =
      ???
  def bar(a: Boolean)(b: T)(c: T):
      T = if (a) b else c
}
```

```scala
inline trait A[T]:
  val x: T = ???
  val y: T = ???
  val z = (x, y)

  def foo(i: Int, j: Double): T =
      ???
  def bar(a: Boolean)(b: T)(c: T):
      T = if a then b else c
```

We then create subclasses for these, following the pattern below:

```scala
class C0 extends A[Boolean]
class C1 extends A[Byte]
class C2 extends A[Char]
// [...]
class C8 extends A[Unit]
class C9 extends A[Boolean]
class C10 extends A[Byte]
// [...]
```

We compare the total size of the class files generated when creating a certain number of children extending the specialized class/inline trait, as well as the time taken to compile the source code.

Results are presented in figures 3 and 4. The compilation time was computed by running the compilation command five times in order to "warm up" the machine, then by computing the mean computation time over five more compilations.

It is however important to note that due to the need to use the modified version of the compiler to compile inline traits, the compilation times for the Scala 3 file were taken from the output of the `scalac` command of sbt, which does not provide decimals. The compilation time for the Scala 2 code was obtained by using the `time` command of bash. Therefore, a difference of about one second between compilation times is not considered to be significant.

First, let us notice that there are only nine types over which a class may be specialized in Scala 2. This means that the code using `@specialized` will generate nine specialized versions of `A`, leading to a fixed cost in space of 29KiB: 22KiB for the specialized classes, and 7KiB for the generic class. This explains the large differences in sizes for less than 10 children classes.

We see that the code duplication induced by the new inlining mechanism leads to a higher cost in space between 10 and 100 children classes. This shows that inline traits are less performant than the Scala 2 annotation when many non-inline classes need to extend them.

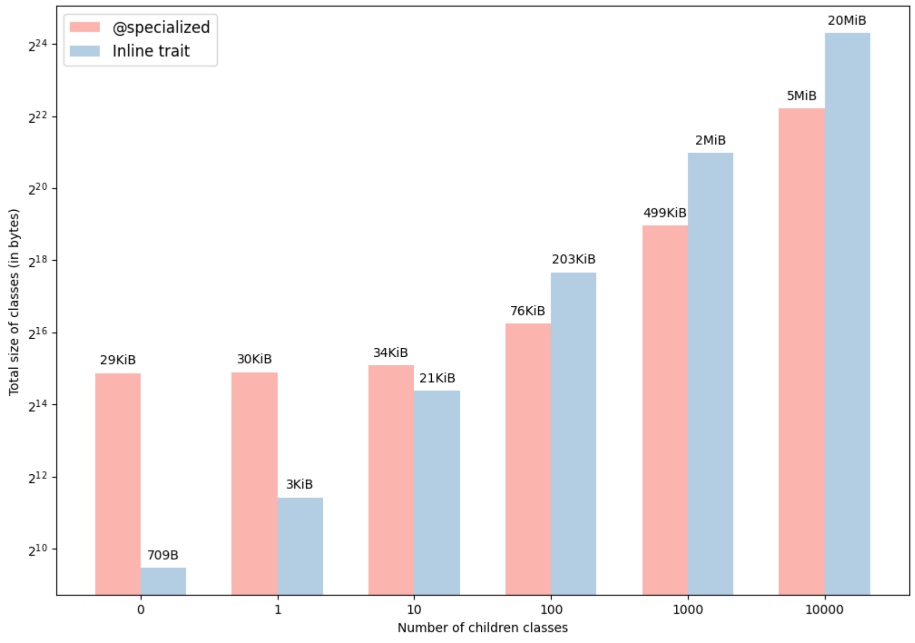

Figure 3: Number of children classes vs. total size of generated class files

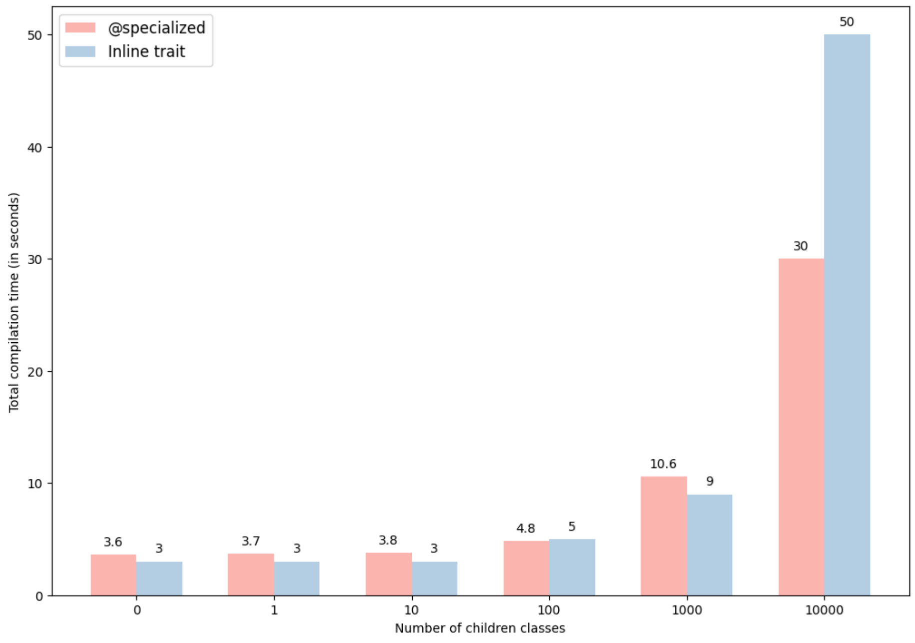

Figure 4: Number of children classes vs. compilation time

Regarding the compilation time, we see that they are comparable when less than a thousand subclasses are involved. However, inline traits seem to require a larger amount of time to compile a very large number of children classes; this is most probably due to the need to inline the trait's code in every child.

Another important value is the time taken by the first warm-up compilation: while the Scala 2 code had compilation times that were always close to the results displayed, the Scala 3 code took much longer for its first compilation: when using 10000 children classes, the first compilation took 108 seconds, as opposed to the 50 seconds presented in the graph. We do not know the precise reason for this behavior.

4.3 Conclusion

We have seen that inline traits are not a perfect replacement for `@specialized`: having a large number of classes extend inline traits leads to an increase both in code generated and time taken to compile in regards to the Scala 2 solution. However, inline traits provide a way to reduce the amount of code generated when they have few extending classes.

In terms of code performance, inline traits provide the same level of improvement that `@specialized` does in Scala 2.

We conclude that inline traits are a good specialization mechanism in Scala 3, when used to inline code in a relatively small number of classes.

5 Future work

Due to time constraints, the implementation is still a proof of concept for inline traits: some elements of the specification are missing, and some corner cases remain unexplored. We will discuss future plans to bring this new feature to a state worthy of being included in dotty.

5.1 Known bugs in the current implementation

Symbols

- the only symbols that are adapted for now are the ones inside of class-likes extending inline traits, this needs to be done for the rest of the code as well (see section 5.2.1 below)

Members

- private members are not renamed, only private parameter accessors are; this needs to be changed so that all overridable private members are renamed

Types

- opaque types are not inlined properly

Inner class-likes

- inner traits' term parameters caused multiple issues, they are forbidden for now
- inner traits cannot have members whose signature include a generic type declared by it (e.g. `class Inner[T](t: T)`)
- inner traits that extend a generic class see their parent tree being wrongfully changed (`Inner extends C[T]` becomes `Inner extends T`)
- inner inner class-likes (a class-like in an inner class-like) are not handled properly; the inlining of inner class-likes need to be separate from the code used to inline the inline trait's body

5.2 Remaining features to implement

5.2.1 New phase *postSpecializeInlineTraits*

For now, the phase *specializeInlineTraits* is responsible for adapting the symbols to point to the ones that have been created and inlined inside of the children class-like. However, we would like for this phase to only be run if there are inline traits to be compiled or inlined, akin to how the *inlining* phase is only run if there are inline methods to compile or method calls to inline.

The implementation should introduce a new phase, which goal would be the following: for each symbol in the code, if it was inlined from an inline trait, replace it with the symbol that was inlined in the non-inline child. For example:

```
inline trait A[T](x: T):
  def foo: T = x

class B extends A[Int](3)

val b = B()

// The symbol foo here does not point to the specialized member of B, but rather
    to the one that it inherits from A, resulting in unboxing after erasure.
    This symbol foo should be replaced with the other symbol foo, that is
    specialized and inlined inside of B, in order to avoid autoboxing.
val x = b.foo
```

5.2.2 Finish supporting inner class-likes

For now, only inner traits are supported by the implementation: the constructors proxies are not yet generated. This would need to be done in two different places:

- in **SpecializeInlineTraits.scala**, in method `transformInlineTrait`, along with the transformation of the inner class-like and the generation of the type,
- in **Inlines.scala**, in `inlinedClassDef`, which would return the inlined inner class-like as well as the implementation of all proxies.

The calls to the inner class-like constructors would need to be replaced everywhere with calls to the constructors proxies.

Inner traits also need to be able to have term and type parameters, which is currently not the case. The implementation of these features must be done in order for inline traits to maximize their potential.

Finally, inner class-likes which signature cannot be transformed into a trait without losing information should be forbidden for now, and an error message should explain this.

5.2.3 Prevent the generation of useless code in inline traits

The compiler still generates some code that is not useful, due to the fact that it considers it to simply be a trait with a modifier. We need to prevent this code from being generated.

An example of this is the generation of an `$init` definition inside inline traits during the *constructors*. Traits usually place their body statements inside of it, and the primary constructor of extending classes simply call this method. However, since the statements of inline traits are directly inlined inside of their children's bodies, this method will always be empty. Both the `$init` definition and calls to it should therefore not be generated.

5.2.4 Miscellaneous changes

Inner class-likes

- private inner class-likes are not treated differently, but they should: they can be inlined like any other non-class-like member

User experience

- print a warning if a member is accessed through an inline trait (see listing 27), and advise the user to change the type to avoid autoboxing
- create more meaningful error messages when inlining goes wrong

5.3 General improvements

5.3.1 Implement postphase checks

Most phases of the compiler have checks in place to ensure that the phase went well, and that no tree is malformed. It would be a great addition to verify that all symbols contain the correct owner and info, and that all trees have the correct type.

5.3.2 Leveraging the power of the inliner even further

Another way of performing the specialization of inline traits was briefly discussed, but due to time constraints, it could not be tried. There is a high chance that it would create less friction with the current implementation of the `Inliner` class, as we currently pass as arguments trees with formats that the class does not expect.

Instead of placing the `@BodyAnnotation` on the inline trait, we would place one such annotation on each of its members that needs to be inlined, and each annotation would only contain the body of the statement it is attached to.

When the specialization phase runs, instead of retrieving the whole body of the parent inline trait and inlining its statements, we would simply create copies of the members inside of the child class-like, with as body a reference to the super member. For example:

```scala
inline trait A[T](x: T):
  def foo(a: Boolean, b: Double)(c: T): T = x
  val y: Char =
    println(x)
    '1'

class B extends A[Int](3):
  // The following members would be generated instead of the ones presented in
      this document:
  override def foo(a: Boolean, b: Double)(c: Int): Int = super.foo(a, b)(c)
  override val y: Char = super.y
```

We would then slightly modify the inliner to also inline code that are is not a method call, as long as the call to `super` is made on a member which has a `@BodyAnnotation`.

It might very well be that this solution would solve some edge cases that currently need to be handled separately by the specializer.

This solution would require a lot of refactoring and rewriting, but we believe that it can be worthy of the time spent implementing it.

References

[1] Matthew H Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library.* Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 9780201309560.

[2] Gilad Bracha, Martin Odersky, and David Stoutamire. “GJ: Extending the JavaTM programming language with type parameters”. In: *Sun Microsystems, University of South Australia, Bell Labs, Lucent Technologies* (1998). URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=705390b032d3263cbd143760e84cad7c74d01a8a (visited on 06/22/2023).

[3] 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. Genova, Italy: Association for Computing Machinery, 2009, pp. 42–47. ISBN: 9781605585413. DOI: 10.1145/1565824.1565830.

[4] *Dropped: Class Shadowing.* URL: https://dotty.epfl.ch/docs/reference/dropped-features/class-shadowing.html (visited on 06/20/2023).

[5] *Explicit (full) template specialization.* URL: https://en.cppreference.com/w/cpp/language/template_specialization (visited on 06/06/2023).

[6] Martin Odersky. *Revive or replace @specialized?* 2022. URL: https://github.com/lampepfl/dotty/issues/15532#issue-1285715806 (visited on 06/12/2023).

[7] Martin Odersky et al. “An Overview of the Scala Programming Language”. In: (2004). URL: https://infoscience.epfl.ch/record/52656 (visited on 06/22/2023).

[8] *Scala Standard Library 2.13.4 - scala.inline.* URL: https://www.scala-lang.org/api/2.13.4/scala/inline.html (visited on 06/12/2023).

[9] *SIP-25 - Trait Parameters.* URL: https://docs.scala-lang.org/sips/trait-parameters.html (visited on 06/15/2023).

[10] Nicolas Stucki et al. “Semantics-Preserving Inlining for Metaprogramming”. In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Scala.* SCALA 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 14–24. ISBN: 9781450381772. DOI: 10.1145/3426426.3428486.

Appendix

A Boxing and unboxing behind the scenes

```scala
class Wrapper[T](val x: T)
class IntWrapper(val x: Int)

val numbers = 1 to 1000000
val ws = numbers.map(Wrapper(_))
val iws = numbers.map(IntWrapper(_))

val noUnboxing =
  iws.map(w => IntWrapper(w.x*w.x + 2*w.x)).foldLeft(0)(_ + _.x)
val withUnboxing =
  ws.map(w => Wrapper(w.x*w.x + 2*w.x)).foldLeft(0)(_ + _.x)
```

Listing 34: Operations on `Int`, with and without autoboxing

A benchmark was run to evaluate the performance of the last two value declarations in listing 34, with the following parameters:

- 5 warm-up iterations of 10 seconds
- 10 measurement iterations of 10 seconds
- benchmark performed 3 times

The results are presented in figure 5.

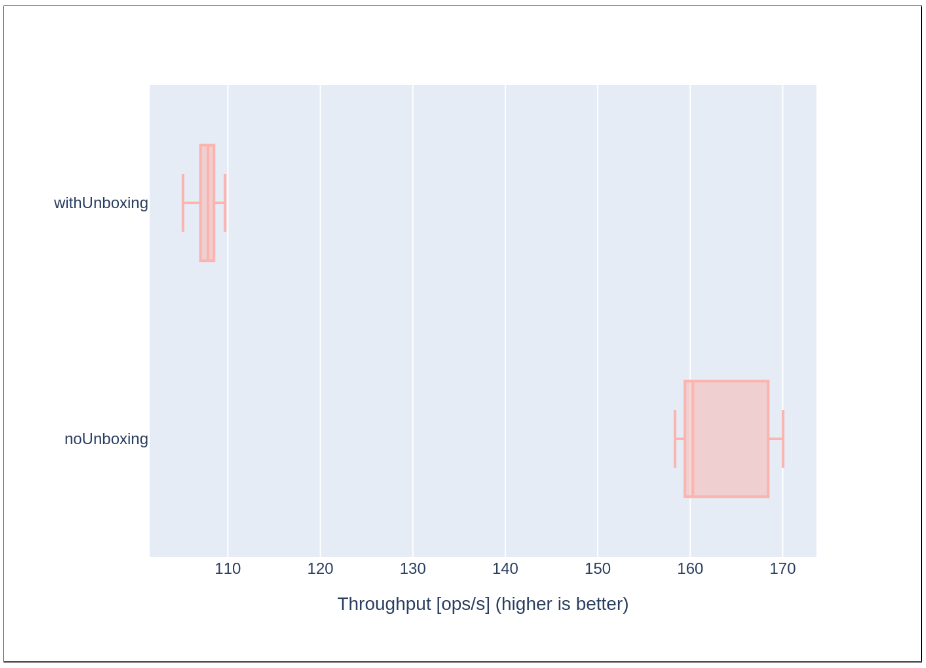

Figure 5: Number of computations per second of the values

We see that boxing and unboxing operations take a noticeable toll on performances, which time-critical code might not be allowed to spare. This shows that there is a need for improvement of performances on generic code, for example through specialization.

B Effect of `@specialize` in Scala 2

B.1 Source code

```scala
object Obj {
  import scala.specialized

  class WrapperNotSpe[T](val x: T)
  // For this example, we specialize only on Int
  class WrapperSpe[@specialized(Int) T](val x: T)

  val wn1 = new WrapperNotSpe(1)
  val wn2 = new WrapperNotSpe(2)
  // Boxing/unboxing will happen
  val wn3 = new WrapperNotSpe(wn1.x + wn2.x)

  val ws1 = new WrapperSpe[Int](1)
  val ws2 = new WrapperSpe[Int](2)
  // Boxing/unboxing will not happen
  val ws3 = new WrapperSpe[Int](ws1.x + ws2.x)

  // Going through a generic environment forces the use of the
  // generic accessors; specialization loses its use here
  def f[T](w: WrapperNotSpe[T]): T = w.x
  def f[T](w: WrapperSpe[T]): T = w.x

  // Both of these values will be unboxed
  val fwn1 = f(wn1)
  val fws1 = f(ws1)
}
```

B.2 Code generated after `specialize` phase

The comments, spacing and indentations in the listing hereafter have been added manually; they are not part of the code generated by the compiler.

All of the code listings below are contained inside the `object Obj`, which is not shown for readability.

B.2.1 `class WrapperNotSpe`

```scala
class WrapperNotSpe[T] extends Object {
  private[this] val x: T = _;
  def x(): T = WrapperNotSpe.this.x;

  def <init>(x: T): Obj.WrapperNotSpe[T] = {
    WrapperNotSpe.super.<init>();
    ()
  }
};
```

B.2.2 `class WrapperSpe` (generic)

```scala
class WrapperSpe[@specialized(scala.Int) T] extends Object {
  // Field x is protected to be accessible by child
  protected[this] val x: T = _;
  def x(): T = WrapperSpe.this.x;

  def <init>(x: T): Obj.WrapperSpe[T] = {
    WrapperSpe.super.<init>();
    ()
  };

  // Bridge method
  def x$mcI$sp(): Int = WrapperSpe.this.x().asInstanceOf[Int]();
  def specInstance$(): Boolean = false
};
```

B.2.3 `class WrapperSpe` (specialized)

```scala
class WrapperSpe$mcI$sp extends Obj.WrapperSpe[Int] {
  // Note that this specialized field replaces entirely the generic one,
  // however the generic field will still be present in the class definition
  protected[this] val x$mcI$sp: Int = _;
  def x$mcI$sp(): Int = WrapperSpe$mcI$sp.this.x$mcI$sp;

  override def x(): Int = WrapperSpe$mcI$sp.this.x$mcI$sp();

  def <init>(x$mcI$sp: Int): Obj.WrapperSpe$mcI$sp = {
    WrapperSpe$mcI$sp.super.<init>(null.asInstanceOf[Int]());
    ()
  };

  def specInstance$(): Boolean = true
};
```

B.2.4 Values declarations

```
private[this] val wn1: Obj.WrapperNotSpe[Int] =
  new Obj.WrapperNotSpe[Int](1);

private[this] val wn2: Obj.WrapperNotSpe[Int] =
  new Obj.WrapperNotSpe[Int](2);

private[this] val wn3: Obj.WrapperNotSpe[Int] =
  new Obj.WrapperNotSpe[Int](
    Obj.this.wn1().x().+(Obj.this.wn2().x())
  );

private[this] val ws1: Obj.WrapperSpe[Int] =
  new Obj.WrapperSpe$mcI$sp(1);

private[this] val ws2: Obj.WrapperSpe[Int] =
  new Obj.WrapperSpe$mcI$sp(2);

private[this] val ws3: Obj.WrapperSpe[Int] =
  new Obj.WrapperSpe$mcI$sp(
    Obj.this.ws1().x$mcI$sp().+(Obj.this.ws2().x$mcI$sp())
  );

private[this] val fwn1: Int = Obj.this.f[Int](Obj.this.wn1());

private[this] val fws1: Int = Obj.this.f[Int](Obj.this.ws1());
```

B.3 Code generated after `erasure` phase

All of the code listings below are contained inside the `object Obj`, which is not shown for readability.

B.3.1 `class WrapperNotSpe`

```
class WrapperNotSpe extends Object {
  private[this] val x: Object = _;
  def x(): Object = WrapperNotSpe.this.x;

  def <init>(x: Object): Obj.WrapperNotSpe = {
    WrapperNotSpe.super.<init>();
    ()
  }
};
```

B.3.2 `class WrapperSpe` (generic)

```
class WrapperSpe extends Object {
  protected[this] val x: Object = _;
  def x(): Object = WrapperSpe.this.x;

  def <init>(x: Object): Obj.WrapperSpe = {
    WrapperSpe.super.<init>();
    ()
  };

  def x$mcI$sp(): Int = unbox(WrapperSpe.this.x());
  def specInstance$(): Boolean = false
};
```

B.3.3 `class WrapperSpe` (specialized)

```
class WrapperSpe$mcI$sp extends Obj.WrapperSpe {
  protected[this] val x$mcI$sp: Int = _;
  def x$mcI$sp(): Int = WrapperSpe$mcI$sp.this.x$mcI$sp;

  override def x(): Int = WrapperSpe$mcI$sp.this.x$mcI$sp();

  def <init>(x$mcI$sp: Int): Obj.WrapperSpe$mcI$sp = {
    WrapperSpe$mcI$sp.super.<init>(null);
    ()
  };

  def specInstance$(): Boolean = true;

  override def x(): Object = scala.Int.box(WrapperSpe$mcI$sp.this.x())
}
```

B.3.4 Values declarations

```
private[this] val wn1: Obj.WrapperNotSpe =
  new Obj.WrapperNotSpe(scala.Int.box(1));

private[this] val wn2: Obj.WrapperNotSpe =
  new Obj.WrapperNotSpe(scala.Int.box(2));

private[this] val wn3: Obj.WrapperNotSpe =
  new Obj.WrapperNotSpe(
    scala.Int.box(
      unbox(Obj.this.wn1().x()).+(unbox(Obj.this.wn2().x()))
    )
  );

private[this] val ws1: Obj.WrapperSpe =
  new Obj.WrapperSpe$mcI$sp(1);

private[this] val ws2: Obj.WrapperSpe =
  new Obj.WrapperSpe$mcI$sp(2);

private[this] val ws3: Obj.WrapperSpe =
  new Obj.WrapperSpe$mcI$sp(
    Obj.this.ws1().x$mcI$sp().+(Obj.this.ws2().x$mcI$sp())
  );

private[this] val fwn1: Int = unbox(Obj.this.f(Obj.this.wn1()));

private[this] val fws1: Int = unbox(Obj.this.f(Obj.this.ws1()));
```

C Normal trait vs. inline trait

The code being compiled is the following, once with the `inline` keyword in the signature of `trait A`, and once without it:

```
inline trait A[T](val x: T):
  def foo: T = x

class B extends A[Int](1)
```

C.1 Without `inline` keyword

```
trait A() extends Object {
  def x(): Object
  def foo(): Object = this.x()
}

class B extends Object, A {
  def <init>(): Unit =
    {
      super()
      this.x = Int.box(1)
      ()
    }
  private val x: Object
  def x(): Object = this.x
  def foo(): Object = super[A].foo()
}
```

C.2 With `inline` keyword

```
inline trait A() extends Object {
  def x(): Object
  def foo(): Object
}

class B extends Object, A {
  def <init>(): Unit =
    {
      super()
      this.x = 1
      ()
    }
  private val x: Int
  override def x(): Int = this.x
  override def foo(): Int = this.x()
  override def x(): Object = Int.box(this.x())
  override def foo(): Object = Int.box(this.foo())
}
```

D Benchmarks

The benchmark tool used is a modified version of Jmh, called *scala3-bench-micro/Jmh*.

D.1 Matrix library benchmark

The following code is used to run the benchmark:

```scala
import org.openjdk.jmh.annotations._
import java.util.concurrent.TimeUnit.SECONDS
import scala.util.Random

@Fork(10)
@Threads(3)
@Warmup(iterations = 3, time = 5, timeUnit = SECONDS)
@Measurement(iterations = 5, time = 10, timeUnit = SECONDS)
@State(Scope.Benchmark)
class MatrixBenchmark {
  val n: Int = 100

  def intMatrixElems: List[List[Int]] =
    List.tabulate(n, n)((_, _) => Random.nextInt())

  @Param(Array("standard", "specialized", "inlinetrait"))
  var libType: String = _

  var m1: BenchmarkMatrix = _
  var m2: BenchmarkMatrix = _

  @Setup(Level.Trial)
  def setup = {
    Random.setSeed(n)

    val matrixFactory = BenchmarkMatrix.ofType(libType)
    m1 = matrixFactory(intMatrixElems)
    m2 = matrixFactory(intMatrixElems)
  }

  @Benchmark
  def matrixBenchmark = (m1 + m2) * m1 // O(n^3) loops
}
```

Listing 35: Benchmark file for matrix operations

```scala
import standard.IntMatrixLib.{Matrix => StdIntMatrix}
import specialized.IntMatrixLib.{Matrix => SpeIntMatrix}
import inlinetrait.IntMatrixLib.{Matrix => InlIntMatrix}

trait BenchmarkMatrix:
  def +(n: BenchmarkMatrix): BenchmarkMatrix
  def *(n: BenchmarkMatrix): BenchmarkMatrix

object BenchmarkMatrix:
  def ofType(tpe: String): Seq[Seq[Int]] => BenchmarkMatrix =
    (elems: Seq[Seq[Int]]) => tpe.toLowerCase() match {
      case "standard" => StdBenchmarkMatrix(StdIntMatrix(elems*))
      case "specialized" => SpeBenchmarkMatrix(SpeIntMatrix(elems*))
      case "inlinetrait" => InlBenchmarkMatrix(InlIntMatrix(elems*))
    }

private class StdBenchmarkMatrix(val m: StdIntMatrix) extends BenchmarkMatrix:
  import standard.IntMatrixLib.{+, `*`}
  override def +(n: BenchmarkMatrix): StdBenchmarkMatrix = n match {
    case stdN: StdBenchmarkMatrix => StdBenchmarkMatrix(this.m + stdN.m)
  }
  override def *(n: BenchmarkMatrix): StdBenchmarkMatrix = n match {
    case stdN: StdBenchmarkMatrix => StdBenchmarkMatrix(this.m * stdN.m)
  }

private class SpeBenchmarkMatrix(val m: SpeIntMatrix) extends BenchmarkMatrix:
  import specialized.IntMatrixLib.{+ => plus, `*` => times}
  override def +(n: BenchmarkMatrix): SpeBenchmarkMatrix = n match {
    case speN: SpeBenchmarkMatrix => SpeBenchmarkMatrix(plus(this.m)(speN.m))
  }
  override def *(n: BenchmarkMatrix): SpeBenchmarkMatrix = n match {
    case speN: SpeBenchmarkMatrix => SpeBenchmarkMatrix(times(this.m)(speN.m))
  }

private class InlBenchmarkMatrix(val m: InlIntMatrix) extends BenchmarkMatrix:
  import inlinetrait.IntMatrixLib.{+, `*`}
  override def +(n: BenchmarkMatrix): InlBenchmarkMatrix = n match {
    case inlN: InlBenchmarkMatrix => InlBenchmarkMatrix(this.m + inlN.m)
  }
  override def *(n: BenchmarkMatrix): InlBenchmarkMatrix = n match {
    case inlN: InlBenchmarkMatrix => InlBenchmarkMatrix(this.m * inlN.m)
  }
```

Listing 36: Common representation of matrices

Hereafter are the three implementations used: *standard*, without specialization; *specialized*, with code similar to the behavior of `@specialized` in Scala 2, and *inlinetrait*, which uses the new construct to specialize the code.

D.1.1 *standard*

```scala
package standard

import scala.reflect.ClassTag

trait MatrixLib[T: ClassTag]:
  opaque type Matrix = Array[Array[T]]

  object Matrix:
    def apply(rows: Seq[T]*): Matrix =
      rows.map(_.toArray).toArray

  extension (m: Matrix)
    def apply(x: Int)(y: Int): T = m(x)(y)
    def rows: Int = m.length
    def cols: Int = m(0).length

object IntMatrixLib extends MatrixLib[Int]:
  extension (m: Matrix)
    def +(n: Matrix): Matrix =
      val sum =
        for row <- 0 until m.rows
        yield
          for col <- 0 until m.cols
          yield m(row)(col) + n(row)(col)
      Matrix(sum*)
    end +

    def *(n: Matrix): Matrix =
      val prod =
        for i <- 0 until m.rows
        yield
          for j <- 0 until n.cols
          yield
            val mults = for k <- 0 until n.rows yield m(i)(k) * n(k)(j)
            mults.fold(0)(_ + _)
      Matrix(prod*)
    end *
```

D.1.2 *specialized*

For readability, the code hereafter is the Scala 2 code that was adapted to work without the `@specialized` annotation. The actual code was manually specialized to work in Scala 3.

```scala
package specialized

import scala.reflect.ClassTag
import scala.specialized

class MatrixLib[@specialized(Int) T: ClassTag] {
  type Matrix = Array[Array[T]]

  object Matrix {
    def apply(rows: Seq[T]*): Matrix =
      rows.map(_.toArray).toArray
  }

  def get(m: Matrix)(x: Int)(y: Int): T = m(x)(y)
  def rows(m: Matrix): Int = m.length
  def cols(m: Matrix): Int = m(0).length
}

object IntMatrixLib extends MatrixLib[Int] {
  def +(m: Matrix)(n: Matrix): Matrix = {
    val sum = {
      for (row <- 0 until rows(m))
        yield {
          for (col <- 0 until cols(m))
            yield m(row)(col) + n(row)(col)
        }
    }
    Matrix(sum: _*)
  }

  def *(m: Matrix)(n: Matrix): Matrix = {
    val prod = {
      for (i <- 0 until rows(m))
        yield {
          for (j <- 0 until cols(n))
            yield {
              val mults =
                for (k <- 0 until rows(n)) yield get(m)(i)(k) * get(n)(k)(j)
              mults.fold(0)(_ + _)
            }
        }
    }
    Matrix(prod: _*)
  }
}
```

D.1.3 *inlinetrait*

For demonstration purposes, the code hereafter does not show the changes made so that the current implementations of inline traits may work. In the actual code, the type `Matrix` is not opaque, and the object `Matrix` is replaced with a method of the same name.

```scala
package inlinetrait

import scala.reflect.ClassTag

inline trait MatrixLib[T: ClassTag]:
  opaque type Matrix = Array[Array[T]]

  object Matrix:
    def apply(rows: Seq[T]*): Matrix =
      rows.map(_.toArray).toArray

  extension (m: Matrix)
    def apply(x: Int)(y: Int): T = m(x)(y)
    def rows: Int = m.length
    def cols: Int = m(0).length

object IntMatrixLib extends MatrixLib[Int]:
  extension (m: Matrix)
    def +(n: Matrix): Matrix =
      val sum =
        for row <- 0 until m.rows
        yield
          for col <- 0 until m.cols
          yield m(row)(col) + n(row)(col)
      Matrix(sum*)
    end +

    def *(n: Matrix): Matrix =
      val prod =
        for i <- 0 until m.rows
        yield
          for j <- 0 until n.cols
          yield
            val mults = for k <- 0 until n.rows yield m(i)(k) * n(k)(j)
            mults.fold(0)(_ + _)
      Matrix(prod*)
    end *
```

D.2 Pairs benchmark

The following code is used to run the benchmark:

```scala
import org.openjdk.jmh.annotations._
import java.util.concurrent.TimeUnit.SECONDS
import scala.util.Random

@Fork(10)
@Threads(3)
@Warmup(iterations = 3, time = 5, timeUnit = SECONDS)
@Measurement(iterations = 5, time = 10, timeUnit = SECONDS)
@State(Scope.Benchmark)
class PairsBenchmark {
  var numPairs: Int = 3_000_000

  def pairElems: List[(First, Second)] = List.tabulate(numPairs)(_ % 2 match {
    case 0 => (Random.nextInt(), Random.nextDouble())
    case 1 => (Random.nextInt(Char.MaxValue).asInstanceOf[Char],
         Random.nextInt(Short.MaxValue).asInstanceOf[Short])
  })

  @Param(Array("standard", "specialized", "inlinetrait"))
  var libType: String = _

  var pairs: List[BenchmarkPair] = _

  @Setup(Level.Trial)
  def setup = {
    Random.setSeed(numPairs)

    val pairFactory = (l: List[(First, Second)]) => l.map((_1, _2) =>
         BenchmarkPair.ofType(libType)(_1, _2))
    pairs = pairFactory(pairElems)
  }

  @Benchmark
  def pairsBenchmark = pairs.foldLeft(0){ case (sum, pair) => pair match {
      case BenchmarkPair(i: Int, d: Double) => 7 * i + 3 * d.toInt + sum
      case BenchmarkPair(c: Char, s: Short) => 5 * c + 2 * s + sum
    }
  }
}
```

Listing 37: Benchmark file for pairs operations

```scala
import standard.IntMatrixLib.{Matrix => StdIntMatrix}
import specialized.IntMatrixLib.{Matrix => SpeIntMatrix}
import inlinetrait.IntMatrixLib.{Matrix => InlIntMatrix}

trait BenchmarkMatrix:
  def +(n: BenchmarkMatrix): BenchmarkMatrix
  def *(n: BenchmarkMatrix): BenchmarkMatrix

object BenchmarkMatrix:
  def ofType(tpe: String): Seq[Seq[Int]] => BenchmarkMatrix =
    (elems: Seq[Seq[Int]]) => tpe.toLowerCase() match {
      case "standard" => StdBenchmarkMatrix(StdIntMatrix(elems*))
      case "specialized" => SpeBenchmarkMatrix(SpeIntMatrix(elems*))
      case "inlinetrait" => InlBenchmarkMatrix(InlIntMatrix(elems*))
    }

private class StdBenchmarkMatrix(val m: StdIntMatrix) extends BenchmarkMatrix:
  import standard.IntMatrixLib.{+, `*`}
  override def +(n: BenchmarkMatrix): StdBenchmarkMatrix = n match {
    case stdN: StdBenchmarkMatrix => StdBenchmarkMatrix(this.m + stdN.m)
  }
  override def *(n: BenchmarkMatrix): StdBenchmarkMatrix = n match {
    case stdN: StdBenchmarkMatrix => StdBenchmarkMatrix(this.m * stdN.m)
  }

private class SpeBenchmarkMatrix(val m: SpeIntMatrix) extends BenchmarkMatrix:
  import specialized.IntMatrixLib.{+ => plus, `*` => times}
  override def +(n: BenchmarkMatrix): SpeBenchmarkMatrix = n match {
    case speN: SpeBenchmarkMatrix => SpeBenchmarkMatrix(plus(this.m)(speN.m))
  }
  override def *(n: BenchmarkMatrix): SpeBenchmarkMatrix = n match {
    case speN: SpeBenchmarkMatrix => SpeBenchmarkMatrix(times(this.m)(speN.m))
  }

private class InlBenchmarkMatrix(val m: InlIntMatrix) extends BenchmarkMatrix:
  import inlinetrait.IntMatrixLib.{+, `*`}
  override def +(n: BenchmarkMatrix): InlBenchmarkMatrix = n match {
    case inlN: InlBenchmarkMatrix => InlBenchmarkMatrix(this.m + inlN.m)
  }
  override def *(n: BenchmarkMatrix): InlBenchmarkMatrix = n match {
    case inlN: InlBenchmarkMatrix => InlBenchmarkMatrix(this.m * inlN.m)
  }
```

Listing 38: Common representation of matrices

Hereafter are the three implementations used: *standard*, without specialization; *specialized*, with code similar to the behavior of `@specialized` in Scala 2, and *inlinetrait*, which uses the new construct to specialize the code.

D.2.1 *standard*

```scala
package standard

class Pair[+T1, +T2](val _1: T1, val _2: T2)
```

D.2.2 *specialized*

The code hereafter is the Scala 2 code that was adapted to work without the `@specialized` annotation. The actual code was manually specialized to work in Scala 3.

```scala
package specialized

import scala.specialized

class Pair[@specialized(Int, Char) +T1, @specialized(Double, Short) +T2](_1: T1,
    _2: T2) {}
```

D.2.3 *inlinetrait*

```scala
package inlinetrait

inline trait Pair[+T1, +T2](val _1: T1, val _2: T2)

class IntDoublePair(override val _1: Int, override val _2: Double) extends
    Pair[Int, Double](_1, _2)
class CharShortPair(override val _1: Char, override val _2: Short) extends
    Pair[Char, Short](_1, _2)
```

D.3 Number of classes generated and total size

Hereafter are the code snippets used to determine the number of class files and their total size when compiling in Scala 2 with `@specialized` (on the left) or in Scala 3 with an inline trait (on the right).

No specialization

```scala
class A[T, U, V] {
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all = (x, y, z)
}
```

```scala
inline trait A[T, U, V]:
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all = (x, y, z)
```

`(Int, U, V)`

```scala
import scala.specialized
class A[@specialized(Int) T, U, V] {
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all = (x, y, z)
}
```

```scala
inline trait A[T, U, V]:
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all: (T, U, V) = (x, y, z)
class C[U, V] extends A[Int, U, V]
```

(Int, Int, Int)

```scala
import scala.specialized
class A[@specialized(Int) T,
    @specialized(Int) U,
    @specialized(Int) V] {
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all = (x, y, z)
}
```

```scala
inline trait A[T, U, V]:
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all: (T, U, V) = (x, y, z)
class C extends A[Int, Int, Int]
```

(Int, Double, Boolean)

```scala
import scala.specialized
class A[@specialized(Int) T,
    @specialized(Double) U,
    @specialized(Boolean) V] {
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all = (x, y, z)
}
```

```scala
inline trait A[T, U, V]:
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all: (T, U, V) = (x, y, z)
class C extends A[Int, Double,
    Boolean]
```

(Int, Double | Int, Boolean)

```scala
import scala.specialized
class A[@specialized(Int) T,
    @specialized(Double, Int) U,
    @specialized(Boolean) V] {
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all = (x, y, z)
}
```

```scala
inline trait A[T, U, V]:
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all: (T, U, V) = (x, y, z)
class C1 extends A[Int, Double,
    Boolean]
class C2 extends A[Int, Int,
    Boolean]
```

(Int, Int, Int) | (Double, Double, Double)

```scala
import scala.specialized
class A[@specialized(Double, Int)
    T, @specialized(Double, Int)
    U, @specialized(Double, Int)
    V] {
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all = (x, y, z)
}
```

```scala
inline trait A[T, U, V]:
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all: (T, U, V) = (x, y, z)
class C1 extends A[Int, Int, Int]
class C2 extends A[Double, Double,
    Double]
```

All primitive types

```scala
import scala.specialized
class A[@specialized T,
    @specialized U, @specialized
    V] {
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all = (x, y, z)
}
```

```scala
inline trait A[T, U, V]:
  val x: T = ???
  val y: U = ???
  val z: V = ???
  def all: (T, U, V) = (x, y, z)
class C1 extends A[Boolean,
    Boolean, Boolean]
class C2 extends A[Boolean,
    Boolean, Byte]
// [...]
class C729 extends A[Unit, Unit,
    Unit]
```