Implementing value classes in Dotty, a compiler for Scala

Author: Guillaume Martres
Supervisor: Martin Odersky

June 23, 2015

Abstract

Classes in Scala are not a free abstraction: on the JVM, every class instance must be a reference to a heap-allocated object, this incurs both CPU and memory overhead. Value classes are a subset of classes which follow a list of restrictions that allow the Scala compiler to reduce this overhead. This is accomplished by having two runtime representations for values: an ”unboxed” one and a ”boxed” one, the latter being only used when necessary. In this report, we explain in details our implementation of value classes in Dotty, an experimental compiler for Scala. We will also mention possible future extensions to the value class mechanism and how they could be implemented.

1 Primitive classes

The ability to create value classes was introduced in Scala 2.10. Before that, only nine primitive value classes existed: `Unit`, `Bool`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float` and `Double`. Since user-defined value classes share many similarities with primitive classes, it is useful to study their design.
Two kinds of values exist on the JVM: nullable references to heap-allocated objects and the following primitive types: `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`. This is done for performance reasons: if the JVM used objects instead of primitives, it would significantly increase its allocation rate, memory usage (because every object has a header) and number of pointer indirections [1].
In Scala, unlike Java, primitives are classes, this simplifies the language and makes it more uniform because users do not have to worry about the distinction between JVM primitives and objects most of the time [2]. This is made possible by having two different runtime representation for a primitive value: when possible we represent it as a JVM primitive, but when the semantics of the JVM require us to provide an object we wrap the primitive in an immutable object containing a single field. We call these two representations *unboxed* and *boxed* respectively.
The following example illustrates the need for boxing:

```scala
val x: Int = 42
val y: AnyVal = x
def identity[T](x: T): T = x
val z: Int = identity(42)
```

On line 2, `x` needs to be boxed because the expected type of the expression is `AnyVal` and primitives on the JVM do not have a common supertype. On line 4, `42` needs to be boxed because generic types are not support by the JVM and need to be erased to their

[1]With a different garbage collection design, other approaches may have been possible, like tagged primitives

[2]Since primitives are not references, `null` is not a valid value and there is no way to do reference equality checks on them: they are *identityless*. This is enforced by making it impossible to call `eq` or `ne` on primitives.

upper bound. Note that values are only boxed when absolutely necessary, for example on line 4 the result of the `identity` method is unboxed so that `z` can be a primitive value:

```scala
val x: Int = 42
val y: Any = scala.Int.box(x)
def identity[T](x: T): T = x
val z: Int = scala.Int.unbox(identity(scala.Int.box(42)))
```

2 User-defined value classes

Scala 2.10 introduced a way for users to create their own value classes [3]. User-defined value classes need to explicitly extend `AnyVal` and to follow some restrictions, for example:

```scala
class Meter(val underlying: Double) extends AnyVal
```

Like primitive classes, this class has two representation: an unboxed representation, `Double`, and a boxed representation, `Meter`. Users can write code using the boxed representation and when possible, the value class transformation will instead use the unboxed representation at runtime.

2.1 Restrictions

The value class transformation is not fully general, it can only be applied to a class `V` if it follows a series of restrictions, most of these restrictions stem from the fact that at any point in the program we may need to box a value class or to unbox it and these operations should not have any side-effect.

`V` has exactly one parameter which is a `val`

This parameter is called the *underlying value* of `V`. It must be a `val` so that the class contains a getter to convert an instance of `V` to its unboxed representation.

`V` must be *ephemeral*

The notion of ephemerality was introduced in SIP-15 [3]. A class or trait `V` is ephemeral if:

- `V` does not declare any field (other than its `val` parameter).
- `V` does not contain any object definition.
- `V` does not have any initialization statement.

`V` may not have secondary constructors

This restriction is not fundamental: we could allow secondary constructors without side-effects but we choose not to do so as that would complicate the value class transformation and users can simply rewrite their code as follow:

```
class Position(val underlying: Long) extends AnyVal {
  def this(posX: Int, posY: Int) =
    this((posX.toLong << 32) + posY.toLong)
}
object Test {
  def test = new Position(10, 20)
}
```

becomes:

```
class Position(val underlying: Long) extends AnyVal
object Position {
  def apply(posX: Int, posY: Int) =
    new Position((posX.toLong << 32) + posY.toLong)
}

object Test {
  def test = Position(10, 20)
}
```

`V` must not define concrete `equals` and `hashCode` methods

This restriction is explained in Section 3.3 and a proposal for lifting it is described in Section 4.2.

`V` must be either a toplevel class or a member of a statically accessible object

If `V` is not statically accessible, then its constructor isn't either and that means that we cannot box it, consider the following example:

```
class Outer {
  class Meter(val underlying: Int) extends AnyVal
}
object Test {
  def getMeter: Outer#Meter = {
    val o = new Outer
    new o.Meter(42)
  }
  def test = {
    val m: Outer#Meter = getMeter
    val a: Any = m
  }
}
```

`Outer#Meter` will be erased to `Int`, so `m` in `test` will have type `Int`, but `m` is then stored in a field of type `Any`: this requires boxing `m` into a `Outer#Meter`, but this is impossible without an instance of `Outer`.

`V` must be `final`

This is needed so that when we unbox a value class then box it back, we always get an instance of the same class, if `V` was not final then boxing would be ambiguous: which subclass constructor should we use? Note that every class that extends `AnyVal` is implicitly `final`, it is not necessary to specify it.

`V` may only extend *universal* traits

Traits in Scala extend `AnyRef` by default, so value classes cannot extend them. Traits which explicitly extend `Any` are called universal, they do not extend `AnyRef` and value classes can extend them. Like value classes and for the same reasons, universal traits need to be ephemeral.

3 Implementation details

3.1 Compiler phases in Dotty

Dotty is split into phases, these phases can transform nodes of the AST (parsed from source files) and they can also transform symbols (inferred from the AST and from classfiles). Dotty is a very modular compiler: as of June 2015, it contains over forty phases. This modularity is possible thanks to the concept of mini-phases. Traditionally, each compiler phase needs to traverse the full AST, but in Dotty phases can be grouped together such that the AST is only traversed once for the full group. As Figure 1 shows, there are only 12 groups of phases in Dotty. Furthermore, note that the phases specific to value classes were inserted into existing groups: they did not require more traversals of the AST. Thus, we were able to make the value class transform mostly self-contained and split into simple pieces without sacrificing performances.

3.2 The value class transformation, step by step

In this section we will describe how each phase of the value class transformation works. We will use the following class as our running example:

```
class Meter(val underlying: Double) extends AnyVal {
  def plus(other: Meter): Meter =
```

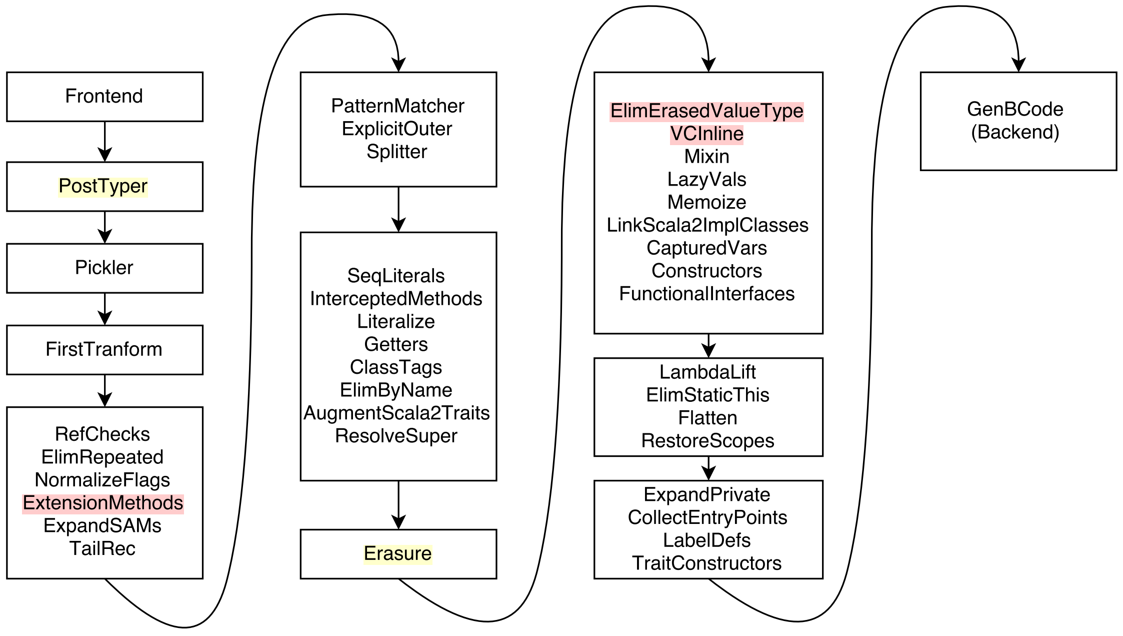

Figure 1: The compiler phases. Each rectangle corresponds to a group of phases. Phases highlighted in red are specific to value classes. Phases highlighted in yellow had to be modified to support value classes.

```
   new Meter(this.underlying + other.underlying)
}
```

3.3 SyntheticMethods

This phase does not appear in Figure 1 because it is actually part of `PostTyper` for performance reasons, but we can reason about it separately from the rest of `PostTyper` anyway. Its original purpose is the addition of the following methods (unless explicitly overridden) in case classes:

```
def equals(other: Any): Boolean
def hashCode(): Int
def canEqual(other: Any): Boolean
def toString(): String
```

To be able to optimize `==` on value classes we need to override `equals` and `hashCode`, so it seems natural to make `SyntheticMethods` responsible for this. Our implementation of these methods must allow us to perform the following peephole optimization (see Section 3.7 for details) for a value class `V`:

```
new V(u1) == new V(u2)
```

should be rewritten to:

```
u1 == u2
```

To figure out how to achieve this, let us first take a look at the specification of `==` [1, Section 12.1]:

```
final def == (that: Any): Boolean =
  if (null eq this) null eq that else this equals that
```

This means that `new V(u1) == new V(u2)` will behave like `(new V(u1)).equals(new V(u2))`, so we just need to define `equals` so that instances of value classes are equal if and only if their underlying value is equal:

```
def equals(that: Any) = that match {
  case that: V => this.underlying == that.underlying
  case _ => false
}
```

This also requires us to define `equals` in a way that will satisfy the requirement that `x == y` implies `x.hashCode == y.hashCode` while still being useful for hashing:

```
def hashCode: Int = this.underlying.hashCode
```

Note that this is different from the default implementations of `equals` and `hashCode` in case classes with one element:

```
def equals(that: Any) =
  (this eq that) || {
    that match {
      case that: V => this.underlying == that.underlying
      case _ => false
    }
  }

def hashCode: Int = {
  var acc: Int = 0xcafebabe
  acc = Statics.this.mix(acc, this.underlying)
  Statics.this.finalizeHash(acc, 1)
}
```

`equals` in case classes has an additional check for referential equality which does not make sense for value classes. `hashCode` is more complicated because it uses the same hashing technique that is used for case classes with several fields. In the future we may decide to use the same `hashCode` implementation for both case classes of one element and value classes, but currently, if a case class is also a value class, `SyntheticMethods` will generate the same `equals` and `hashCode` methods as if it was a non-case value class.

3.4 ExtensionMethods

The methods defined in a value class `V` can only be called on an instance of `V`. Our first step in making it possible to avoid allocation is to make it possible to call these methods statically. For every method `m` defined in the body of `V`:

1. We create a new method `m$extension` whose first parameter list is `$this: V` and whose remaining parameter lists are the parameter lists of `m`. The body of `m$extension` is a copy of the body of `m` where every implicit and explicit reference to `this` has been replaced by `$this`. This new method is called an *extension method*.

2. We replace the body of `m` by a forwarder to the corresponding extension method.

For example, given a method like `plus` in `Meter`:

```
def plus(other: Meter): Meter =
  new Meter(this.underlying + other.underlying)
```

We add the following definition to the companion object `Meter`:

```
def plus$extension($this: Meter)(other: Meter): Meter =
  new Meter($this.underlying + other.underlying)
```

And we replace `plus` in the class `Meter` by a forwarder:

```
def plus(other: Meter): Meter =
  Meter.plus$extension(this)(other)
```

3.4.1 Handling of overloaded methods

When two overloaded methods `m` in a value class `V` exist, instead of creating two overloaded extension methods `m$extension`, we create `m$extension1` and `m$extension2`. We do not maintain an explicit mapping between each overloaded method and the corresponding extension method. Instead, given a symbol for an instance method `m`, `ExtensionMethods#extensionMethod` will find all extension methods whose name start with `m$extension` and return the only one whose signature agrees with the signature of `m`. This pattern is used because overloaded resolution is a complex operation that should ideally only be necessary during the initial typer phase of the compiler to avoid surprising behavior.

3.4.2 Phase placement

This phase should be placed after every phase that may add methods to a value class (like `SyntheticMethods`), otherwise these methods won't get a corresponding extension method and calling them will require boxing. It should also preferably be before phases which may significantly increase the size of methods like `PatternMatcher` since we need to traverse and transform the body of every value class method to generate the extension methods.

3.5 Erasure

The purpose of `Erasure` is to translate Scala types to something representable on the JVM. This is a rather complex operation because the Scala type system contains concepts (like the unification of primitive types and reference types, or generic arrays) which are not easily expressible on the JVM. Most of this complexity is contained in `Erasure#Typer` which is a subclass of `Typer`, the class used to type trees in the frontend. Extending `Typer` allows us to reuse the following features in erasure:

Expected type When typing a tree node, it is useful to know its expected type. For example when typing `val x: Foo = if (cond) a else b` the expected type of `a` is `Foo`. In `Erasure`, when the computed type of a node is not a subtype of its expected type, we need to perform *type adaptation* which will be explained in the next section.

Type propagation Once a node has been typed, its type should be propagated upward in the tree to keep things consistent. For example in the tree `if (cond) a else b` the computed type is the least-upper-bound of the types of the nodes `a` and `b`.

Before describing how `Erasure` works for value classes we will explain how it works for primitives. Both cases deal with similar problems (primitives and value classes both have an unboxed and a boxed representation on the JVM) but erasing primitives is simpler.

3.5.1 Primitive erasure

We will use the primitive class `Int` for all of our examples but every primitive (except `Unit`, which we won't describe) is erased in the same way.

```scala
val x: Int = 42
val y: Any = x
def identity[T](x: T): T = x
val z: Int = identity(42)
```

Conceptually, we start by just erasing the types:

- The primitive class `Int` is erased to the JVM primitive type `int` [3].

[3] In practice, erased trees still contain the class type `Int` but the logic in `Erasure` treats this type as if

- `Any`, which is a superclass of every Scala class is erased to `Object`, which is a superclass of every JVM class.
- Since the JVM does not support generic methods, the type parameter `T` of `identity` has to be erased to its upper bound `Any` (`[T]` is equivalent to `[T <: Any]`) which is subsequently erased to `Object`.

The result is:

```
val x: int = 42
val y: Object = x
def identity(x: Object): Object = x
val z: int = identity(42)
```

At this point, our code no longer typechecks: on line 2 for example, the computed type of `x` is `int` but its expected type is `Object`, we need to perform type adaptation. The type adaptation rules for primitives are fairly simple, using `int` as an example:

- `e` becomes `scala.Int.box(e)` if `e` has type `int` and its expected type is not `int`
- `e` becomes `scala.Int.unbox(e)` if the expected type of `e` is `int` but its actual type is not `int`

After applying these rules, our tree typechecks again:

```
val x: int = 42
val y: Object = scala.Int.box(x)
def identity(x: Object): Object = x
val z: int = scala.Int.unbox(identity(scala.Int.box(42)))
```

3.5.2 Value Class erasure

We will start by considering the following simple example:

```
val m: Meter = new Meter(3)
```

We would like to erase `Meter` to its erased underlying type `double`, so let's try replacing `Meter` by `double`:

```
val m: double = new Meter(3)
```

As with primitives, we need to use type adaptation to make this typecheck. For a value class `V` whose single field is called `underlying` we can simply adapt `e` to `e.underlying` when the type of `e` is `V` and the expected type is `double`:

it was a primitive type. In this report, we write `int` in Scala code to mean "`Int` as it is treated in `Erasure` and afterwards", this is similar to the meaning of `int` in [7].

```
val m: double = new Meter(3).underlying
```

So far this works, but what if we add the following line:

```
val a: Any = m
```

Before `Erasure` this is valid because `Meter` is a subtype of `Any`. But after erasing the types, we get:

```
val a: Object = m
```

At this point we need a type adaptation rule to box `m` into a `Meter`, but we cannot write this rule: the type of `m` is `double` and its expected type is `Object`, so the rule for primitive boxing from the previous section applies and we get:

```
val a: Object = scala.Double.box(m)
```

This is **not** what we want: a boxed `Double` and a boxed `Meter` are different even though their unboxed representations are the same, the problem is that we lost information when we erased `Meter` to `double`.
To resolve this we need to use an intermediate representation whose only purpose is to drive type adaptation: for a value class `V` whose underlying type is `U` this type is called `ErasedValueType(V, U)`. It is outside the normal hierarchy of Scala types: it is a subtype of no other type and is a supertype only of `Nothing`. Instead of replacing `V` by `U` we will replace it by `ErasedValueType(V, U)`:

```
val m: ErasedValueType(Meter, double) = new Meter(3)
val a: Object = m
```

Before presenting the type adaptation rules to make this tree typecheck we need to discuss two special methods that exist for every value class:

```
object V {
  def u2evt$(x0: U): ErasedValueType(V, U)
  def evt2u$(x0: ErasedValueType(V, U)): U
}
```

For every value class `V` we create symbols for these methods in `ExtensionMethods` but we do not create any corresponding tree: the only purpose of these methods is to cast values to and from `ErasedValueType`. These methods may seem unnecessary: in Scala we can cast any value to any type using `asInstanceOf`. But this is not the case after `Erasure`: only values whose type is a reference type can be casted using `asInstanceOf`.
We can now introduce the correct type adaptation rules for value classes:

- `e` is adapted to `V.u2evt$(e.underlying)` if `e` has type `V` and the expected type is

`ErasedValueType(V, U)`

- `e` is adapted to `new V(V.evt2u$(e))` if the type of `e` is `ErasedValueType(V, U)` and its expected type is any other type

After applying these rules, our tree typechecks again, we have successfully erased value classes:

```
val m: ErasedValueType(Meter, Int) =
  Meter.u2evt$(new Meter(3).underlying)
val a: Object = new Meter(Meter.evt2u$(m))
```

3.6 ElimErasedValueType

After `Erasure`, `ErasedValueType` is no longer needed: we will not do more type adaptations. Therefore, we can safely remove every use of `evt2u$` and `u2evt$` and replace `ErasedValueType(V,U)` by `U` in every type, this is the purpose of `ErasedValueType`. This phase will also remove the symbols `V.evt2u$` and `V.u2evt$` from the scope they were entered in since they should not be used in any subsequent phase.

3.7 VCInline

Before describing what `VCInline` does, let us take a look at what we have accomplished so far. We are able to translate the following code:

```
val m1: Meter = new Meter(1)
val m2: Meter = new Meter(2)
val m3: Meter = m1.plus(m2)
```

to this:

```
val m1: double = new Meter(1).underlying
val m2: double = new Meter(2).underlying
val m3: double = new Meter(m1).plus(new Meter(m2))
```

At first glance, it may look like we haven't made any step towards solving our original problem: we now have more allocations than at the start. But we have established an important property: every `val`, `var`, method parameter or method return value whose type was `Meter` in the source code is now an `double`. This means that we only call value class constructors in two cases:

- When the JVM type system requires us to use a boxed representation. This occurs when we upcast a value class to one of its supertype because these supertypes do not have one unboxed representation (for example, `class Foo(i: Int) extends AnyVal` and `class Bar(d: Double) extends AnyVal` both have `AnyVal` as a supertype but their unboxed representations do not have any common supertype on the JVM), so we have to use a boxed representation. In this case the value class transformation will not be able to avoid allocations.

- When we access the single field of the class or when we call a class method. In both of these cases we can avoid allocation by using a few simple rewriting rules: this is what `VCInline` does.

The goal of this phase is to elide value class constructor calls when possible. This is safe because we disallow initialization statements as well as `val` and `var` fields inside value classes, so constructor calls cannot have any side-effect.
For every value class `V` with a field `underlying`, this phase performs the following peephole optimizations:

- `v.m(args)` where `v` is an expression of type `V` and `m` is a method defined in `V` is rewritten as `V.m$extension(v.underlying(), args)`
- `(new V(u)).underlying()` is rewritten as `u`.
- `new V(u1) == new V(u2)` is rewritten as `u1 == u2`. This is guaranteed to be correct because of the definitions of `equals` and `hashCode` in `SyntheticMethods`, see Section 3.3.

4 Future work and possible improvements

4.1 Value classes without explicit annotation

We require value classes to explicitely extend `AnyVal`, but the compiler could automatically detect that a class follows the value class restrictions and treat it as if it was a true value class. The main reason for not doing this is that making a class into a value class is a binary incompatible change and it would be very surprising for users of the language if a seemingly innocuous change to a class would break binary compatibility. However, this optimization would be valid under a closed world assumption and could be done by an optional linker phase for Dotty.

4.2 Allow overriding `equals` and `hashCode` in value classes and universal traits

Currently, these two methods cannot be redefined by the user and are always synthesized in `SyntheticMethods` (Section 3.3). This allows us to perform the following optimization in `VCInline` (Section 3.7):

```
new V(u1) == new V(u2)
```

is rewritten as:

```
u1 == u2
```

We cannot just disable this optimization if a value class or one of the universal trait it extends redefines `equals` or `hashCode` because that would mean that adding or removing these methods from value classes would be a binary-incompatible change. Instead, following a proposition by Dmitry Petrashko, we add the following overload of `==` to every value class `V`:

```
// If V and its supertraits do not redefine equals nor hashCode:
def == (that: V): Boolean = this.underlying == that.underlying
// Otherwise:
def == (that: V): Boolean = this.equals(that)
```

Because of the value class transformation, having this overload in the class will result in:

```
new V(u1) == new V(u2)
```

being rewritten as:

```
V.==$extension(u1, u2)
```

When neither `equals` nor `hashCode` are redefined this is as efficient as our original optimization: the body of `V.==$extension` will simply compare the underlying values without boxing, otherwise it will box and delegate the equality check to `equals`.

4.3 Value classes and specialization

Dotty will soon support type parameter specialization [2, Chapter 4]. Since value classes can contain type parameters the value class transformation may need to be adapted to work well with the specialization transformation. In the current prototype for method specialization [6] the transformation is done by a new phase `TypeSpecializer` placed between `Splitter` and `SeqLiterals` (see Figure 1). This phase creates new specialized methods to avoid boxing, for example:

```
def identity[@specialized(Int) T](x: T): T = x
val x: Int = identity(42)
```

becomes after `TypeSpecializer`:

```
def identity[@specialized(Int) T](x: T): T = x
def identity$mIc$sp(x: Int): Int = x
val x: Int = identity$mIc$sp(42)
```

A new specialized method is created and calls to the generic method are replaced by calls to the specialized method when appropriate. To understand how this affect value classes, let us see what happens with the following code:

```scala
class Foo(val underlying: Int) extends AnyVal {
  def foo[@specialized(Int) T](x: T): T = x
}
val x: Int = new Foo(1).foo[Int](2)
```

after `ExtensionMethods` this becomes:

```scala
class Foo(val underlying: Int) extends AnyVal {
  def foo[@specialized(Int) T](x: T): T = Foo.foo$extension[T](this)(x)
}
object Foo {
  def foo$extension[@specialized(Int) T]($this: Foo)(x: T): T = x
}
val x: Int = new Foo(1).foo[Int](2)
```

and after `TypeSpecializer` we get:

```scala
class Foo(val underlying: Int) extends AnyVal {
  def foo[@specialized(Int) T](x: T): T = Foo.foo$extension[T](this)(x)
  def foo$mIc$sp(x: Int): Int = Foo.foo$extension$mIc$sp[T](this)(x)
}
object Foo {
  def foo$extension[@specialized(Int) T]($this: Foo)(x: T): T = x
  def foo$extension$mIc$sp($this: Foo)(x: Int): Int = x
}
val x: Int = new Foo(1).foo$mIc$sp(2)
```

We would like to rewrite the last line to:

```scala
Foo.foo$extension$mIc$sp(1, 2)
```

But the current implementation of `VCInline` cannot do that: it relies on `ExtensionMethods#extensionMethod`. For a method `m`, `extensionMethod` will simply look for all the extension methods whose names start with `m$extension` and pick the one whose signature agrees with the signature of `m`, but in our case we have a method named `foo$mIc$sp` whose extension method is called `foo$extension$mIc$sp`, so it won't be found by `extensionMethod`. Below we discuss two potential ways of solving this issue, but more work is needed to determine what the final solution will be.

4.3.1 Add an API to navigate between generic and specialized methods

For example, if we had the following methods available:

```scala
def specializedClassParams(specializedMeth: Symbol): List[Type]
def specializedMethodParams(specializedMeth: Symbol): List[Type]
def genericMethod(specializedMeth: Symbol): Symbol
def specializedMethod(genericMeth: Symbol,
  classParams: List[Type], methParams: List[Type]): Symbol
```

then finding the correct specialized extension method should be easy:

```
def specializedExtensionMethod(meth: Symbol): Symbol = {
  val classParams = specializedClassParams(meth)
  val methParams = specializedMethParams(meth)
  val genericMeth = genericMethod(meth)
  val genericExtensionMeth =
    ExtensionMethods.extensionMethod(genericMeth)
  specializedMethod(genericExtensionMeth, classParams, methParams)
}
```

4.3.2 Inline value class calls into extension methods calls before `TypeSpecialization`

An alternative would be to move the `VCInline` rewriting rule for extension methods to its own phase which would be run just before `TypeSpecialization`, we could call it `VCInlineCalls`. The tree for our previous example would look like this after `VCInlineCalls`:

```
val x: Int = Foo.foo$extension[Int](new Foo(1), 2)
```

after `TypeSpecializer` this becomes:

```
val x: Int = Foo.foo$extension$mIc$sp(new Foo(1), 2)
```

and finally, after `VCInline` no allocation remain:

```
val x: Int = Foo.foo$extension$mIc$sp(1, 2)
```

Unfortunately, doing the rewriting before `Erasure` seems challenging when we have class type parameters:

```
class Bar[T](val underlying: Int) extends AnyVal {
  def bar: Int = 42
}
// e is an expression of type Bar
val x: Int = e.bar
```

We would like to rewrite `e.bar` to `Bar.bar$extension[e.T](e)`, but this is difficult for two reasons:

- `T` is not accessible from outside the body of `Bar`, we will need to disable accessibility checks.
- `e` might not be a stable path [1, Section 3.1], in this case we will need to rewrite `e.bar` to:

```
{
  val v = e
  Bar.bar$extension[v.T](v)
}
```

4.4 Extension methods in universal traits

In our implementation of value classes, we do not generate extension methods for universal traits methods. This means that calling one of these methods on a value class requires an allocation. Universal extension methods are more complex to implement than class extension methods because a universal trait may be extended by value classes with different underlying types: we need our extension methods to be polymorphic without incurring any boxing overhead. In this section, we describe a scheme for achieving this using class specialization, this scheme is based on Dmitry Petrashko's proposal [4]. We will illustrate it using the following example:

```
trait Foo extends Any {
  def foo = 0
  def print = {
    println(this.foo)
    println(this)
  }
}
class Bar(val underlying: Int) extends Foo {
  def foo = this.underlying
}
val x = (new Bar(1)).print
```

The universal extension methods of `Foo` will be stored in an *extension trait* called `Foo$extension`. We need to use a trait instead of a companion object so that we can override universal extension methods: for example, the companion object `Bar` will extend `Foo$extension` and override `foo$extension`. Previously, `$this` in extension methods had the same type as `this` in the corresponding instance method and was only erased to its underlying type in `Erasure`, but for universal traits no such erasure is possible since they cannot be erased to one specific underlying type, so we need a new representation. For universal traits, the type of `$this` is `T`, a trait type parameter which represent the underlying type of the current value class. Instead of replacing `this` by `$this` in extension methods, we replace `this.m(...)` by `m$extension($this, ...)`, and when a boxed representation of this is needed, we replace it by `box$($this)`, where `box$` is an abstract method in each universal trait and defined in every value class. To be able to override universal extension methods by class extension methods we need to use the same representation of `$this` for class extension methods too. For example, the companion object `Bar` will extend `Foo$extension[Int]` (because the underlying type of `Bar` is `Int`), so `$this` will have type `T=Int` in the companion object `Bar`.

```
trait Foo$extension[@specialized T] {
  def box$($this: T): Foo
  def foo$extension($this: T) = 0
  def print$extension($this: T) = {
    println(foo$extension($this))
    println(box$($this))
  }
}
object Bar extends Foo$extension[Int] {
  override def box$($this: Int): Bar = new Bar($this)
  override def foo$extension($this: Int) = (new Meter($this)).underlying
}
```

We will not discuss in details the next steps because they depend on whether `TypeSpecializer` is run before or after the inlining of value class calls, which is still an open question (see Section 4.3). But in both cases, the end result should be that an expression like `(new Bar(3)).print` will be translated to `Bar.print$mIc$sp(3)` which achieves our goal of avoiding boxing when calling a method defined in a universal trait.

4.5 Incompatible type aliases

Ideally, value classes should never have any runtime cost, but because we require them to follow the semantics of regular classes this is not the case. For example, it seems that we should be able to represent `Array[Meter]` as an `Array[int]` at runtime using the same mechanism that allows us to represent `Meter` as `int` at runtime, unfortunately the two representations are not equivalent:

```
def genArray[T](arr: Array[T]) = {
  arr[0] match {
    case _: Meter => println("I contain a Meter")
    case _ =>
  }
}
val x = Array(new Meter(1))
genArray(x) // Should print "I contain a Meter"
```

Scala allows us to determine the type of a value at runtime using pattern matching and `isInstanceOf`, this is useful but it also means that we cannot easily change the runtime representation of values. We could represent `x` as an `Array[int]` and only convert it to `Array[Meter]` when we pass it to a generic method, but having method calls take $\mathcal{O}(n)$ operations would be very surprising for users. What we really need is the ability to define a new type which is always erased to some other type. Scala already has a notion of type aliases:

```
type Meter = Int
```

Here, `Meter` is a type alias of `Int`. It is useful to denote semantics but it does not enforce any rule at compile-time (because `Meter <: Int` and `Int <: Meter`) or run-time (because it will be erased to `Int`). We could introduce a new kind of type aliases such that neither `Meter <: Int` or `Int <: Meter` are true but keep the property that `Meter` erases to `Int`. This concept closely ressembles `newtype` in Haskell, so let's use that keyword:

```
newtype Meter = Int
```

Since `Meter` and `Int` are different types before `Erasure`, the only way to convert between them is to use `asInstanceOf`. To be more type safe let us introduce two new methods to do the conversion: `asAlias` and `dealias`, for example:

```
val x: Meter = 3.asAlias[Meter]
val y: Int = x.dealias
```

This way, `Meter` and `Int` are distinct types but we can still convert values between them if we ask explicitly. Our new `Meter` type does not seem very useful: the only method we can call on it is `dealias`, but we can make it behave like a value class easily by using an implicit value class:

```
implicit class MeterOps(val self: Meter) extends AnyVal {
  def plus(other: Meter): Meter =
    (self.dealias + other.dealias).asAlias[Meter]
  ...
}
val m1: Meter = 1.asAlias[Meter]
val m2: Meter = 2.asAlias[Meter]
val m3: Meter = m1.plus(m2)
```

`MeterOps` will enrich `Meter` with new methods such as `plus`, and since it is a value class these enrichments will not require any allocation. Furthermore, since we never store a value of type `MeterOps` into a field or pass it to a method, we do not have to worry about value class boxing, so we get the best of both worlds.

However, `newtype` will probably not be added to the Scala language because it's a new feature which overlaps with an existing one (value classes) and using it requires knowledge about details of erasure that we would rather not burden the user with. Furthermore, the JVM may end up getting support for real value types in the future [5] at which point `newtype` would become mostly useless.

4.6 Value classes with multiple underlying fields

An implementation of value classes which supports multiple fields has already been demonstrated [7]. However, their implementation required numerous rewriting rules and it did not attempt to work around the JVM limitation of only one return value (their implementation always boxes when returning a value class instance). So it is not yet clear whether or not our implementation will be extended to support multiple fields.

5 Conclusion

In this report we have presented our implementation of value classes in Dotty. By itself, our implementation is interesting because it is rather simple and integrates well into the existing compiler architecture, but we have also tried to make our report detailed enough to showcase how a feature gets integrated into a modern compiler. We proposed some ways of extending our design and discussed how these extensions would increase its complexity. Finally, we note that value classes in Scala can be seen as a "workaround" for the lack of value types on the JVM, and since value types are coming to the JVM [5], we need to plan how this will affect our current implementation and possible future improvements.

References

[1] Martin Odersky et al. *Scala Language Specification*. http://www.scala-lang.org/files/archive/spec/2.11/. 2015.

[2] Iulian Dragos. *Compiling Scala for Performance*. http://library.epfl.ch/theses/?nr=4820. 2010.

[3] Martin Odersky, Jeff Olson, Paul Phillips, and Joshua Suereth. *SIP-15: Value Classes*. http://docs.scala-lang.org/sips/completed/value-classes.html. Feb. 2012.

[4] Dmitry Petrashko. *Proposal for universal extension methods*. https://github.com/lampepfl/dotty/issues/629#issuecomment-108080561. June 2015.

[5] John Rose, Brian Goetz, and Guy Steele. *State of the Values*. http://cr.openjdk.java.net/~jrose/values/values-0.html. Apr. 2014.

[6] Alexandre Sikiaridis. *Implement method type specialisation*. https://github.com/lampepfl/dotty/pull/630. June 2015.

[7] Vlad Ureche, Eugene Burmako, and Martin Odersky. “Late data layout: unifying data representation transformations”. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. ACM. 2014, pp. 397–416. URL: https://github.com/miniboxing/miniboxing-plugin/blob/wip/docs/2014-08-ldl-oopsla.pdf?raw=true.