Implementing Value Classes in Dotty, a compiler for Scala

Author: Guillaume Martres
Doctoral Assistant: Dmitry Petrashko
Supervisor: Martin Odersky

EPFL

Table of Contents

Table of Contents

What is Dotty?

- An experimental compiler for Scala developed at LAMP
- Check it out on github.com/lampepfl/dotty
- The backend (bytecode emission) is (mostly) shared with Scala 2.12.
- Breaks compatibility (a rewriting tool is currently in development)
- Introduces new features
  - Union types: `val x: Int | String = if (foo) 42 else "str"`
  - Trait constructor parameters: `trait T(x: Int) { ... }`
  - More to come once the compiler is more stable.
- Current status: compiles itself, but the resulting code does not run correctly yet.

Dotty phases

- 40 phases currently and more to come
- "Miniphases" are grouped together
- A full retraversal of the AST is only needed for each group
- The value class transform in Dotty is split into several miniphases for modularity and ease of understanding but none of them required the creation of a new group
- This is in contrast with the traditional Scala compiler where each new phase requires a full retraversal of the AST, leading to fewer and bigger phases.

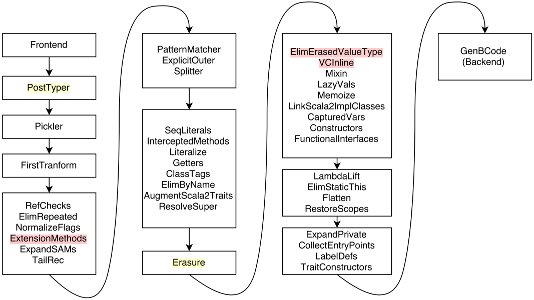

Table of Contents

Quick overview of Value Classes

```
class Meter
```

Quick overview of Value Classes

```
class Meter(val underlying: Int)
```

Quick overview of Value Classes

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

Quick overview of Value Classes

```scala
class Meter(val underlying: Int) extends AnyVal {
  // No initialization statements
```

Quick overview of Value Classes

```
class Meter(val underlying: Int) extends AnyVal {
  // No initialization statements
  // No equals or hashCode methods
```

Quick overview of Value Classes

```scala
class Meter(val underlying: Int) extends AnyVal {
  // No initialization statements
  // No equals or hashCode methods
  def plus(other: Meter): Meter =
     new Meter(this.underlying + other.underlying)
}
```

Goals of Value Classes

- Same semantics as regular classes
- But their runtime representation is "unboxed" when possible:
  - Write this: `val m: Meter = new Meter(1)`
  - Get this at runtime: `val m: Int = 1`
- Benefits: decreased GC pressure, increased memory locality, ...
- Drawbacks: boxed representation sometines needed to preserve semantics:
  - When casting a value class to one of its supertype
  - In arrays of value classes
  - When calling a method defined in a parent of the value class

Table of Contents

SyntheticMethods

- Original purpose: generate the following methods (unless they already exist) in case classes: `equals`, `hashCode`, `canEqual`, `toString`, `productArity`, `productPrefix`
- For value classes, we simply make `equals` and `hashCode` forward to the underlying values:

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

ExtensionMethods

- A class method cannot be called without an instance of a class
- First step in avoiding that: move the body of every method `m` to the companion object of the class to make it static.
- The body of `m` itself is replaced by a forwarder to `m$extension`

```
// Before
class Meter(val underlying: Int) extends AnyVal {
  def plus(other: Meter): Meter =
     new Meter(this.underlying + other.underlying)
}
```

ExtensionMethods

- A class method cannot be called without an instance of a class
- First step in avoiding that: move the body of every method `m` to the companion object of the class to make it static.
- The body of `m` itself is replaced by a forwarder to `m$extension`

```
// After
class Meter(val underlying: Int) extends AnyVal {
  def plus(other: Meter): Meter =
    Meter.plus$extension(this)(other)
}
object Meter {
  def plus$extension($this: Meter)(other: Meter): Meter =
    new Meter($this.underlying + other.underlying)
}
```

Erasure

- Many steps, one of the most complex phase. Cannot be easily separated into miniphases.
- Translate types from the Scala type system to something representable on the JVM.
- This sometimes require **adapting terms**.
- Examples:
  - `List[String]` is erased to `List`
  - Type parameters are erased to their upper-bound
  - `Any`, the supertype of all Scala types, is erased to `Object`, the supertype of all reference types on the JVM.

Erasing primitives

- Replace `Int` by `int` in types

User code	After Erasure
`val m: Int = 3`	`val m: int = 3`

Erasing primitives

- Replace `Int` by `int` in types

User code	After Erasure
`val m: Int = 3` `val box: Any = m`	`val m: int = 3` `val box: Object = m // Type mismatch`

Erasing primitives

- Replace `Int` by `int` in types
- Adapt `Int`-typed tree to `Any` using rule `x -> scala.Int.box(x)`

User code	After Erasure
`val m: Int = 3` `val box: Any = m`	`val m: int = 3` `val box: Object = scala.Int.box(m)`

Erasing value classes: first attempt

- Replace `Meter` by `int` in types

User code	After Erasure
`val m: Meter = new Meter(3)`	`val m: int = new Meter(3)` `// Type mismatch`

Erasing value classes: first attempt

- Replace `Meter` by `int` in types
- Adapt `Meter`-typed tree to `int` using rule `x -> x.underlying`

User code	After Erasure
`val m: Meter = new Meter(3)`	`val m: int = new Meter(3).underlying`

Erasing value classes: first attempt

- Replace `Meter` by `int` in types
- Adapt `Meter`-typed tree to `int` using rule `x -> x.underlying`

User code	After Erasure
`val m: Meter = new Meter(3)` `val box: Any = m`	`val m: int = new Meter(3).underlying` `val box: Object = m // Type mismatch`

Erasing value classes: first attempt

- Replace `Meter` by `int` in types
- Adapt `Meter`-typed tree to `int` using rule `x -> x.underlying`
- Adapt `int`-typed tree to `Any` using rule `x -> ???`

User code	After Erasure
`val m: Meter = new Meter(3)` `val box: Any = m`	`val m: int = new Meter(3).underlying` `val box: Object = m // Type mismatch`

Erasing value classes: the proper way

- New type only used internally by the compiler: `ErasedValueType(V, U)`
- Abbreviated to `EVT(V, U)` to save space on slides
- Replace `Meter` by `EVT(Meter, int)` in types

User code	After Erasure
`val m: Meter = new Meter(3)`	`val m: EVT(Meter, int) = new Meter(3)` `  // Type mismatch`

Erasing value classes: the proper way

- New type only used internally by the compiler: `ErasedValueType(V, U)`
- Abbreviated to `EVT(V, U)` to save space on slides
- Replace `Meter` by `EVT(Meter, int)` in types
- Adapt `Meter` to `EVT(Meter, int)` using rule `x -> x.underlying.asInstanceOf[EVT(Meter, int)]`

User code	After Erasure
`val m: Meter = new Meter(3)`	`val m: EVT(Meter, int) = new Meter(3).underlying .asInstanceOf[EVT(Meter, int)]`

Erasing value classes: the proper way

- New type only used internally by the compiler: `ErasedValueType(V, U)`
- Abbreviated to `EVT(V, U)` to save space on slides
- Replace `Meter` by `EVT(Meter, int)` in types
- Adapt `Meter` to `EVT(Meter, int)` using rule `x -> x.underlying.asInstanceOf[EVT(Meter, int)]`

User code	After Erasure
`val m: Meter = new Meter(3)`	`val m: EVT(Meter, int) = new Meter(3).underlying .asInstanceOf[EVT(Meter, int)]`
`val box: Any = m`	`val box: Object = m // Type mismatch`

Erasing value classes: the proper way

- New type only used internally by the compiler: `ErasedValueType(V, U)`
- Abbreviated to `EVT(V, U)` to save space on slides
- Replace `Meter` by `EVT(Meter, int)` in types
- Adapt `Meter` to `EVT(Meter, int)` using rule `x -> x.underlying.asInstanceOf[EVT(Meter, int)]`
- Adapt `EVT(Meter, int)` to **any other type** using rule `x -> new Meter(x.asInstanceOf[int])`

User code	After Erasure

`val m: Meter = new Meter(3)`	`val m: EVT(Meter, int) =`
	`  new Meter(3).underlying`
	`    .asInstanceOf[EVT(Meter, Int)]`

`val box: Any = m`	`val box: Object =` `  new Meter(m.asInstanceOf[Int])`

ElimErasedValueType

- Replace `EVT(Meter, Int)` by `Int`
- Remove casts that are no longer necessary
- That's it!

User code	After ElimErasedValueType
`val m: Meter = new Meter(3)` `val box: Any = m`	`val m: int = new Meter(3).underlying` `val box: Object = new Meter(m)`

Current state

User code	After ElimErasedValueType
`val m1: Meter = new Meter(3)`	`val m1: int =` `  new Meter(3).underlying`
`val m2: Meter = new Meter(4)`	`val m2: int =` `  new Meter(4).underlying`
`val m3: Meter = m1.plus(m2)`	`val m3: int =` `  new Meter(m1).plus(m2)`
`val b = m1.==(m2)`	`val b =` `  new Meter(m1).==(new Meter(m2))`
`val box: Any = m1`	`val box: Object = new Meter(m1)`

VCInline

- For every value class `V` with a field `underlying`, perform the following peephole optimizations that avoid allocation:
  - For every method `m` declared in `V`, `new V(a).m(b)` becomes `V.m$extension(a, b)`
  - `new V(a).underlying` becomes `a`
  - `new V(a).==(new V(b))` becomes `a.==(b)`

User code	After VCInline
`val m1: Meter = new Meter(3)`	`val m1: Int = 3`
`val m2: Meter = new Meter(4)`	`val m2: Int = 4`
`val m3: Meter = m1.plus(m2)`	`val m3: Int = Meter.plus$extension(m1, m2)`
`val b = m1.==(m2)`	`val b = m1.==(m2)`
`val box: Any = m1`	`val box: Object = new Meter(m1)`

Table of Contents

Overriding `equals` in value classes

- **Problem**: If we allow the user to override `equals` in a value class, the optimization of == is no longer coherent.
- **Solution in Scala 2.x**: Disallow user-defined `equals`
- **Alternative 1**: Disable the == optimization if the user defines `equals`
  - Adding or removing `equals` would be a binary-incompatible change.
- **Alternative 2**: Instead of having a special optimization for ==, treat it like a normal value class method so that we can rewrite:

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

as:

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

Overriding `equals` in value classes

- Definition of == in `Any`

```
final def ==(that: Any): Boolean
```

Overriding `equals` in value classes

- Definition of == in `Any`

```
final def ==(that: Any): Boolean
```

- If the user did not define `equals` in `V` or one of its supertrait:

```
def ==(that: V): Boolean = this.underlying == that.underlying
// The corresponding extension method will be:
def ==$extension($this: U)(that: U): Boolean = $this == that
```

Overriding `equals` in value classes

- Definition of == in `Any`

```
final def ==(that: Any): Boolean
```

- If the user did not define `equals` in `V` or one of its supertrait:

```
def ==(that: V): Boolean = this.underlying == that.underlying
// The corresponding extension method will be:
def ==$extension($this: U)(that: U): Boolean = $this == that
```

- Otherwise:

```
def ==(that: V): Boolean = this.equals(that)
// The corresponding extension method will be:
def ==$extension($this: U)(that: U): Boolean =
  V.equals$extension($this)(new V(that))
```

Example of a non-specialized method

```
def identity[T](x: T): T = x
val x: Int = identity(1)
```

will be erased to:

```
def identity(x: Object): Object = x
val x: int = scala.Int.unbox(identity(scala.Int.box(1)))
```

Example of a specialized method

```
def identity[@specialized(Int) T](x: T): T = x
val x: Int = identity(1)
```

will be erased to:

```
def identity(x: Object): Object = x
def identity$mIc$sp(x: int): int = x
val x: int = identity$mIc$sp(1)
```

Source code

```
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

```
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)
```

After TypeSpecializer

```
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(this)(x)
}
object Foo {
  def foo$extension[@specialized(Int) T]($this: Foo)(x: T): T = x
  def foo$extension$mIc$sp($this: Foo)(x: T): T = x
}
val x: Int = new Foo(1).foo$mIc$sp(2)
// How do we know that the extension method corresponding to
// ‘foo$mIc$sp‘ is called ‘foo$extension$mIc$sp‘ ?
```

Inlining value class methods before Erasure

- To avoid dealing with complex name mangling, we split VCInline into two phases:
  - VCInlineMethods (before Erasure) replaces value class method calls by calls to extension methods
  - VCInlineAllocations (after Erasure) handles the other roles of `VCInline` as before:
    - `new V(a).underlying` becomes `a`
    - `new V(a).==(new V(b))` becomes `a.==(b)`
    - ~~For every method `m` declared in `V`, `new V(a).m(b)` becomes `V.m$extension(a, b)`~~

Example of rewriting done by VCInlineMethods

```
class Bar[T](val underlying: Int) extends AnyVal {
  def bar[A]: Int = 42
}
val x: Int = e.bar[String]
```

- If `e` is a stable prefix, we can rewrite this expression as

  ```
  val x: Int = Bar.bar$extension[String, e.T](e)
  ```

- Otherwise, we need to evaluate it:

  ```
  val x: Int = {
    val v = e
    Bar.bar$extension[String, v.T](v)
  }
  ```

Arrays of unboxed value classes

- Ideally, we would like to erase `Array[Meter]` to `Array[Int]`, but this would prevent us from using arrays of value classes in a generic position.
- However, we could take advantage of the fact that generic arrays in Scala already require runtime support because they're not directly supported by the JVM, for example:

```
def foo[T](arr: Array[T]): Unit = {
  val elem = arr[0]
  arr[1] = elem
}
```

is erased to:

```
def foo(arr: Object): Unit = {
  val elem = ScalaRunTime.array_apply(arr, 0)
  ScalaRunTime.array_update(arr, 1, elem)
}
```

Arrays of unboxed value classes

- Prototype available at github.com/lampepfl/dotty/pull/729

```
val am: Array[Meter] = Array(new Meter(1), new Meter(2))
am[0] = new Meter(3)
val m: Meter = arr[1]
foo(arr)
```

would be erased to:

```
object MeterBoxUnbox extends IntBoxUnbox {
  def box(u: Int): Any = new Meter(u)
  def unbox(b: Any): Int = b.asInstanceOf[Meter].underlying
}
val am: VCIntArray = new VCIntArray(Array(1, 2), MeterBoxUnbox)
am.arr[0] = 3
val m: int = am.arr[1]
foo(arr)
```

Arrays of unboxed value classes

```
final class VCIntArray(val arr: Array[Int], val bu: IntBoxUnbox)
  extends VCArrayPrototype {
  override def apply(idx: Int) =
    bu.box(arr(idx))
  override def update(idx: Int, elem: Any) =
    arr(idx) = bu.unbox(elem)
  override def length: Int = arr.length
  //...
}
trait IntBoxUnbox {
  def box(u: Int): Any
  def unbox(b: Any): Int
}
```

Arrays of unboxed value classes

```scala
abstract class VCArrayPrototype extends Object {
  def apply(idx: Int): Object
  def update(idx: Int, elem: Any): Unit
  def length: Int
}
```

Arrays of unboxed value classes

```
def array_apply(xs: AnyRef, idx: Int): Any = {
  xs match {
    case x: Array[AnyRef] => x(idx).asInstanceOf[Any]
    case x: Array[Int]    => x(idx).asInstanceOf[Any]
    case x: Array[Double] => x(idx).asInstanceOf[Any]
    case x: Array[Long]   => x(idx).asInstanceOf[Any]
    case x: Array[Float]  => x(idx).asInstanceOf[Any]
    case x: Array[Char]   => x(idx).asInstanceOf[Any]
    case x: Array[Byte]   => x(idx).asInstanceOf[Any]
    case x: Array[Short]  => x(idx).asInstanceOf[Any]
    case x: Array[Boolean] => x(idx).asInstanceOf[Any]
    case x: Array[Unit]   => x(idx).asInstanceOf[Any]
    case x: VCArrayPrototype => x.apply(idx)
    case null => throw new NullPointerException
  }
}
```