Opaque types

Understanding SIP-35

Erik Osheim @d6

stripe

who am i?

— typelevel member λ
— maintain spire, cats, and other scala libraries
— interested in expressiveness and performance ◒
— ml-infra at stripe

code at: http://github.com/non

disclaimer

1. This talk discusses potential changes to Scala.
2. The SIP has not yet been approved.
3. It's still possible the proposal will change.

Caveat emptor!

overview

What we will cover:

1. Types, classes, and type aliases
2. SIP-35: opaque types
3. Motivation and various examples
4. Pros, cons, and comparisons

Types, classes, and type aliases

types

What do you think Scala types are?

types

What do you think Scala types are?

— methods of declaring memory locations
— tags attached to values at runtime
— java classes
— sets of values
— systems for constraining values
— things we write after the colon (e.g. x: Int)

types

What do you think Scala types are?

- ~~methods of declaring memory locations~~
- ~~tags attached to values at runtime~~
- java classes
- sets of values
- systems for constraining values
- things we write after the colon (e.g. x: Int)

are types just classes?

String is definitely a type, and also a class.

Maybe this is right?

are types just classes?

Consider the following code:

```scala
class Pair[A](first: A, second: A)
```

It definitely produces a single class:

```
$ scalac Pair.scala && ls -1
Pair.class
Pair.scala
```

are types just classes?

Q: How many distinct types does Pair produce?

```
class Pair[A](first: A, second: A)
```

are types just classes?

Q: How many distinct types does Pair produce?

```
class Pair[A](first: A, second: A)
```

A: Trick question!

```
Pair[Int], Pair[String], Pair[List[Double]],
Pair[Pair[Boolean]], and so on...
```

Given any type T we can produce a new Pair[T].

digression

Pair is a "parameterized type"

- also known as a "type constructor"
- given a type, it produces a type
- sometimes written informally as: `* -> *`
- not a "proper type", it needs a parameter

types

What do you think Scala types are?

- ~~methods of declaring memory locations~~
- ~~tags attached to values at runtime~~
- ~~java classes~~
- sets of values
- systems for constraining values
- things we write after the colon (e.g. x: Int)

sets of values?

```
sealed trait Duprass
sealed trait Minton
case object Horlick extends Minton with Duprass
case object Claire extends Minton with Duprass

val xs: List[Duprass] = List(Horlick, Claire)

val ys: List[Minton] = xs
// <console>:13: error: type mismatch;
//  found   : List[Duprass]
//  required: List[Minton]
//        val ys: List[Minton] = xs
//                               ^
```

types

What do you think Scala types are?

- ~~methods of declaring memory locations~~
- ~~tags attached to values at runtime~~
- ~~java classes~~
- ~~sets of values~~
- systems for constraining values
- things we write after the colon (e.g. x: Int)

type aliases

Type aliases allow us to rename a type:

```
type TrueOrFalse = Boolean

val t1: TrueOrFalse = true
val t2: Boolean     = t1 // ok
val t3: TrueOrFalse = t2 // also ok
```

Notice TrueOrFalse is the same as Boolean.

type aliases

Type aliases can also introduce type constructors:

```scala
type AlwaysInt[A] = Int

type LeftOrRight[A] = Either[A, A]

type AssocRow[K, V] = List[(K, V)]
```

type aliases

```scala
// toy example for illustration

case class User(uid: Long, gid: Long, name: String)

object Db {
  def findById(u: Long): Option[User] = ...
  def findByGroup(g: Long): List[User] = ...
}

val Some(root) = Db.findById(0)
```

type aliases

```scala
type UID = Long
type GID = Long
case class User(uid: UID, gid: GID, name: String)

object Db {
  def findById(u: UID): Option[User] = ...
  def findByGroup(g: GID): List[User] = ...
}

val Some(root) = Db.findById(0) // still works
```

type aliases

```scala
type UID = Long
type GID = Long
case class User(uid: UID, gid: GID, name: String)

object Db {
  def findById(u: UID): Option[User] = ...
  def findByGroup(g: GID): List[User] = ...
}

val Some(root) = Db.findById(0) // still works
val weird = Db.findById(root.gid) // huh?
```

type aliases

Type aliases:

— do not introduce new types
— are completely erased at compile-time
— can introduce type constructors
— can also adapt existing type constructors

SIP-35
opaque types

what's a SIP?

— stands for Scala Improvement Process
— formal proposal to change Scala
— specifies changes to Scala Language Specification
— also includes motivation, examples, etc.
— process has existed since 2012
— rebooted by Scala Center in mid-2016.

sip-35

Co-authored by Jorge Vicente Cantero and Erik Osheim

TL;DR:

```
> This is a proposal to introduce syntax
> for type aliases that only exist at
> compile time and emulate wrapper types.
```

https://docs.scala-lang.org/sips/opaque-types.html

(This document is still evolving, will likely change.)

what does sip-35 mean?

It's easiest to compare opaque types with type aliases.

Type aliases are transparent:

— code can "see through" type aliases in proper types
— authors can inline aliases present in proper types
— aliases do not introduce new types
— are completely erased before runtime
— do not produce classes

what does sip-35 mean?

Opaque types are... well... opaque:

— code cannot see through an opaque type
— authors cannot inline opaque types
— opaque types do introduce new types
— are still completely erased before runtime
— still do not produce classes

let's take a look!

Here's an opaque type to go along with our earlier example:

```
opaque type UID = Long
```

That's it!

well... maybe not

How do you produce a value of type UID?

```scala
opaque type UID = Long

val u1: UID = 0L                         // fails
val u2: UID = new UID(0L)                // nope
val u3: UID = UID(0L)                    // still no
val u4: UID = 0L.asInstanceOf[UID] // cheater! :P
```

location is everything!

How do you produce a value of type UID?

```
opaque type UID = Long

object UID {
  val u1: UID = 0L // ok
}

val u2: UID = 0L // not ok
```

location is everything!

- opaque types may have companion objects
- within this companion opaque types are transparent
- constructors, accesors, and extractors must go there
- otherwise, no access is permitted

what is erasure?

Consider the following:

```
val lst: List[Any] = List(1, "two", 3.0)
lst.foreach(println)
// 1
// two
// 3.0
```

We used toString and println to "recover" type information from lst.

what is erasure?

However, opaque types are different:

```
List(UID.u1, 1.0, "two").foreach(println)
// 0
// 1.0
// two

List(0L, 1.0, "two").foreach(println)
// 0
// 1.0
// two
```

what is erasure?

- Erasure "erases" type information
- UID and Long are indistinguishable at runtime
- opaque types cannot override methods (e.g. toString)

Motivation and various examples

motivation

1. introduce types without classes.
2. give authors more control over erasure.
3. predictable runtime representation/performance
4. limit access to existing classes/types

example: safe nullable

Code as-written by author:

```
opaque type Safe[A <: AnyRef] = A

object Safe {
  def apply[A <: AnyRef](a: A): Safe[A] = a

  def recover[A <: AnyRef](na: Safe[A], a: A): A =
    if (na == null) a else na

  def bind[B <: AnyRef](na: Safe[A],
                        f: A => Safe[B]): Safe[B] =
    if (na == null) null else f(na)
}
```

example: safe nullable

Code as-emitted by compiler:

```
object Safe {
  def apply[A <: AnyRef](a: A): A = a

  def recover[A <: AnyRef](na: A, a: A): A =
    if (na == null) a else na

  def bind[B <: AnyRef](na: A, f: A => B): B =
    if (na == null) null else f(na)
}
```

example: safe nullable

Code as-written by author:

```
val x: Safe[String] = Safe(unsafeJavaApi(...))
val s: String = Safe.recover(x, "")
```

Code as-compiled (post-inlining):

```
val x: String = unsafeJavaApi(...)
val s: String = if (x == null) "" else x
```

That's pretty much the "lowest level" code possible.

example: safe nullable

Differences between `Safe` and `Option`:

— Safe[String] is equivalent to String at runtime
— Safe(...) does not allocate instances, unlike Option(...)
— AnyRef constraint means Safe has no monad
— Safe[Safe[String]] does not type-check
— Safe does not have any methods defined
— modulo-inlining, Safe does not add overhead

example: safe nullable, enriched

```
opaque type Safe[A <: AnyRef] = A

object Safe {
  def apply[A <: AnyRef](a: A): Safe[A] = a

  implicit class Ops[A <: AnyRef](na: Safe[A])
      extends AnyVal {
    def recover(a: A): A =
      if (na == null) a else na
  }
}
```

example: safe nullable, enriched

Code as-written by author:

```
val x = Safe(unsafeJavaApi(...)).recover(a)
val y = Safe(otherApi(...)).recover(b)
f(x, y)
```

Code as-compiled (post-inlining):

```
val x = { val na = unsafeJavaApi(...)
          if (na == null) a else na }
val y = { val nb = otherApi(...)
          if (nb == null) b else nb }
f(x, y)
```

example: safe nullable, enriched

Q: Are the previous inlinings realistic?

A: We think so (more or less):

— methods like apply and recover are very small

— companion's methods are static, should inline well

— enrichment is where value classes work best

— opaque types' constraints allow optimization

example: type tagging

```scala
import scala.{specialized => sp}

// S @@ T means that type S is tagged with tag T
opaque type @@[S, T] = S

object @@ {
  def tag[@sp S, T](s: S): S @@ T = s
  def untag[@sp S, T](st: S @@ T): S = st
  def deepTag[F[_], @sp S, T](fs: F[S]): F[S @@ T] = fs
  def deepUntag[F[_], @sp S, T](fst: F[S @@ T]): F[S] = fst

  implicit def ord[S, T](implicit ev: Ordering[S]): Ordering[S @@ T] =
    deepTag[Ordering, S, T](ev)
}
```

example: type tagging

```scala
import Tagged._
trait Meters
trait Feet

val x: Double @@ Meters = @@.tag[Double, Meters](30.0)
val y: Double @@ Meters = @@.tag[Double, Meters](12.5)
List(x, y).sorted    // ok: List(12.5, 30.0)

val z: Double @@ Feet = @@.tag[Double, Feet](1.0)
List(z, z).sorted    // ok: List(1.0, 1.0)
List(x, y, z).sorted // fails, no Ordering[Any]
```

example: type tagging

Code as-compiled (post-inlining):

```
object @@ {
  def tag[@sp S, T](s: S): S = s
  def untag[@sp S, T](st: S): S = st

  def deepTag[F[_], @sp S, T](fs: F[S]): F[S] = fs
  def deepUntag[F[_], @sp S, T](fst: F[S]): F[S] = fst

  implicit def ord[@sp S, T](implicit ev: Ordering[S]): Ordering[S] =
    ev
}
```

example: type tagging

Code as-compiled (post-inlining):

```
val x: Double = 30.0
val y: Double = 12.5
List(x, y).sorted    // ok: List(12.5, 30.0)

val z: Double = 1.0
List(z, z).sorted    // ok: List(1.0, 1.0)
List(x, y, z).sorted // fails, as shown before
```

reasoning about erasure

Opaque types are opaque at compile-time.

But you can determine their runtime form:

— replace the LHS of an opaque type with its RHS
— inline methods from companion marked @inline
— that's it!
— (optional: inline all "simple" methods in companion)

reasoning about erasure

You can also run this logic in reverse:

- start with some "raw" code
- determine where you wish to limit access
- (or where you wish to improve the type guarantees)
- introduce opaque types there
- add methods to companion as necessary

reasoning about erasure

We often say that opaque types minimize boxing.

This is true but a better formulation might be:

```
> Opaque types do not introducing any boxing
> not already present in the underlying code.
```

example: integer flags

```scala
opaque type Mode = Int

object Mode {
  val Forbidden: Mode = 0
  val Execute: Mode = 1
  val Write: Mode = 2
  val Read: Mode = 4

  implicit class Ops(val lhs: Mode) extends AnyVal {
    def &(rhs: Mode): Mode = lhs & rhs
    def |(rhs: Mode): Mode = lhs | rhs
    def toInt: Int = lhs
  }
}
```

example: integer flags

```
// invalid integers are impossible
// no Option, parsing, error-checking, etc.
val permissions = Mode.Read | Mode.Execute

// could support these methods directly in
// Mode companion instead of using .toInt
grantUnixAccess(permissions.toInt, ...)
```

example: immutable arrays

```
opaque type IArray[A] = Array[A]

object IArray {
  @inline final def init[@sp A](body: => Array[A]): IArray[A] =
    body
  @inline final def size[@sp A](ia: IArray[A]): Int =
    ia.length
  @inline final def get[@sp A](ia: IArray[A], i: Int): A =
    ia(i)
}
```

example: immutable arrays

Code as-written by author:

```
val xs: IArray[Long] = IArray.init { javaApi(...) }
var i: Int = 0
while (i < IArray.size(xs)) {
  val x: Long = IArray.get(i)
  ...
  i += 1
}
```

Notice that xs cannot be mutated.

example: immutable arrays

Code as-emitted by compiler:

```
val xs: Array[Long] = { javaApi(...) }
var i: Int = 0
while (i < xs.length) {
  val x: Long = xs(i)
  ...
  i += 1
}
```

This will operate on long[] and long as hoped.

Pros, cons, and comparisons

what about value classes?

Value classes were introduced in 2.10:

- defined with `extends AnyVal`
- very specific class requirements
- can only extend universal traits
- avoids allocating objects in some cases
- intended to support zero-cost enrichment
- class still exists at runtime

what about value classes?

Value classes have capabilities opaque types lack:

- able to define methods
- can be distinguished from underlying type at runtime
- can participate in subtyping relationships
- can override `.toString` and other methods

what about value classes?

However, value classes have some down sides too:

— unpredictable boxing
— constructor/accessor available by default
— cannot take advantage of specialization
— always allocates when used with arrays
— always allocates when used in a generic context

By contrast, opaque types are always erased.

value class boxing example

Here's a simple value class:

```
class S(val string: String) extends AnyVal {
  def toLower: String = string.toLowerCase
}
```

We want S to be compiled to String when possible.

value class boxing example

When will S be treated as String? When will it box?

```
val s = new S("hi mom") // ok
new S("HI MOM").toLower // ok
class T(x: S)           // ok, `x` field is a String
val t = new T(s)        // ok

val pair = (s, s)       // boxes :/
val arr = Array(s, s)   // boxes :(
val lst = List(s, s)    // boxes :/
val p: S => Boolean =   // will box when called
  (s: S) => s.string.isEmpty
p(s)                    // boxes :P
```

opaque types unboxing example

Here's the same type as an opaque type:

```scala
opaque type S = String

object S {
  def apply(str: String): S = str

  implicit class Ops(val s: S) extends AnyVal {
    def string: String = s
    def toLower: String = s.toLowerCase
  }
}
```

opaque types unboxing example

S will always be treated as a String:

```
val s = S("hi mom") // ok
S("HI MOM").toLower // ok
class T(x: S)       // ok, `x` field is a String
val t = new T(s)    // ok

val pair = (s, s)         // ok, (String, String)
val arr = Array(s, s)     // ok, Array[String]
val lst = List(s, s)      // ok, List[String]
val p: S => Boolean =     // ok, Function1[S, Boolean]
  (s: S) => s.string.isEmpty
p(s)                      // ok
```

when to use value classes?

Value classes are best used:

— to provide low-cost enrichment
— in cases where traditional wrappers are needed
— in direct contexts (e.g. fields/transient values)

(In other cases, value classes may be more marginal.)

opaque type pros

Opaque types:

— work well with arrays
— work well with specialization
— avoid an "abstraction penalty"
— are useful for "subsetting" a type
— offer pleasing minimalism

opaque type cons

However, opaque types also:

— require lots of boilerplate (especially wrappers)
— require a class anyway when doing enrichments
— do not act like traditional classes
— do not eliminate standard primitive boxing
— cannot participate in subtyping

conclusion

SIP-35 is moving quickly!

— Good feedback from last SIP meeting
— We're revising the SIP text
— Jorge continues to work on implemention.
— We're targeting Scala 2.13.

the end

Are you excited about SIP-35? Skeptical? Confused?

Let us know what you think!

Questions, use cases, and comments are very welcome!

Thanks!