

# Opaque Types To Infinity

Stephen Compall

2018-06-04 Mon

# Fetch these slides

`https://is.gd/OpaqueTypesInfinity or`

`http://nocandysw.com/opaque-types-to-infinity.pdf`

# About me

Those slide URLs again: <https://is.gd/OpaqueTypesInfinity> or <http://nocandysw.com/opaque-types-to-infinity.pdf>

...

- I like types
- Scalaz contributor since 2012
- Contributor to Typelevel.scala blog since 2013
- Working at Digital Asset
  - distributed ledger technology (blockchain)
  - typed, purely functional smart contract language in Scala and Haskell (daml.com)
  - Come join us! [digitalasset.com/careers](https://digitalasset.com/careers)

# Goals of newtypes

```
opaque type Label = String
```

- like alias, same runtime representation as RHS (right-hand side) of =
- *unlike* alias, treated as distinct type at compile-time
- *coercible*: compiler-enforced abstraction without wrapping/unwrapping of individual values

## Original inspiration

Julian Michael, mail to scala-user, 19 August 2016,  
<https://is.gd/LowerCaseString>

*Hi all,*

*I might be exploring a well-worn field, but I'll share my whole journey below anyway and list some questions at the end.*

*...*

*It seems to me that you can use abstract types to do opaque sealing to make a type alias that represents a subtype corresponding to an property—say, being in the range of `_.toLowerCase`.*

The ML family of programming languages has all the answers.

## Step 1: declare signature

```
// The SIGNATURE
sealed abstract class LabelModule {
  type Label
  // Put the public *interface* here
  def apply(s: String): Label
  def unwrap(l: Label): String
}
```

## Step 2: define structure, assign to global val

```
// The global VAL
// its singleton type is a "lookup key" to find the real type
val Instance: LabelModule =
  // The STRUCTURE
  new LabelModule {
    type Label = String
    // Put the private *implementation* here
    override def apply(s: String): Label = s
    override def unwrap(l: Label): String = l
  }
```

## Label is an “existential type”

```
import label1.LabelModule.Instance
type Label = Instance.Label
```

```
scala> "hi": Label
error: type mismatch;
 found   : String("hi")
 required: Label
      (which expands to)  label1.LabelModule.Instance.Label
```

```
scala> Instance.apply("hi"): String
error: type mismatch;
 found   : label1.LabelModule.Instance.Label
 required: String
```



# Who said it was a reference type?

```
scala> null: Label
error: type mismatch;
 found   : Null(null)
 required: Label
```

# The existential can be “copied”

```
val Instance2: LabelModule = Instance
```

`Instance.Label` and `Instance2.Label` are incompatible types.

## Two rules to remember

- 1 What is in the *signature* (abstract class or trait) will be seen by outside code, what is in the *structure* (new) will not; that's why the type's RHS only occurs in latter.
- 2 **Always ascribe the val!** If you don't, you'll break the abstraction by revealing the type's RHS.

# How to map a list in $O(1)$

```
// add declaration to signature,  
// implementation to structure  
def wrapList(xs: List[String]): List[Label] = xs
```

```
scala> Instance.wrapList(List("hi", "there"))  
res3: List[label1.LabelModule.Instance.Label]  
    = List(hi, there)
```

# How to map unmappable things in $O(1)$

```
def subst[F[_]](fs: F[String]): F[Label] = fs

// defined in signature; subst is its own inverse
def unsubst[F[_]](fl: F[Label]): F[String] = {
  type K[A] = F[A] => F[String]
  subst[K](identity)(fl)
}

scala> Instance.subst[List](List("hi", "there"))
res4: List[label1.LabelModule.Instance.Label]
     = List(hi, there)
```

# subst is proof of type equality

But it only arises when you ask for it.

```
implicit val labelMonoid: Monoid[Label] =  
  Instance.subst(Monoid[String])
```

No other code, not even the implementation of `Monoid[String]`, knows that `Label = String`.

## But... subst is proof of type equality

What if you don't want to reveal that fact in your signature?

## Suddenly, I care about the conformance relation

In the *signature*,

```
// Label autowidens to String
```

```
type Label <: String
```

```
// String autowidens to Label
```

```
type Label >: String
```

```
// or use both!
```

```
type Label >: String <: CharSequence
```

These tend to be called *translucent* newtypes.



# Translucency has serious implications

```
type Label >: String // in the signature...
```

```
scala> val lbls = List("hi", "there"): List[Label]
lbls: List[translucentlabel.LabelModule.Instance.Label]
      = List(hi, there)
```

# You've never had more power to construct imaginary lattices

```
sealed abstract class LanguagesModule {  
  type Language <: String  
  type Functional <: Language  
  type WellTyped <: Language  
  type SinglyTyped <: Language  
  type TheScalazDream >: Scala.type with Haskell.type  
  val Scala: Functional with WellTyped  
  val Haskell: Functional with WellTyped  
  val JavaScript: SinglyTyped  
}
```

# The cost of subtyping is inheritance

If I write

```
type Label <: String
```

- No way to add a Label-specific reverse method
- No way to add any other method *name* String already uses

If this is a problem, you *must* change or remove your upper bound.  
(Lower >: bounds are not subject to this problem.)

## An incoherent instance

```
sealed abstract class DualModule {
  type Dual[+A] <: A
  def apply[A](a: A): Dual[A]
}

val Instance: DualModule = new DualModule {
  type Dual[+A] = A
  override def apply[A](a: A) = a
}

import Instance.Dual
implicit def dualSemigroup[A](implicit A: Semigroup[A])
  : Semigroup[Dual[A]] =
  Semigroup instance ((l, r) => Instance(A.append(r, l)))
```

# Subtyping is incompatible with disagreeing instances

Or, makes those instances **incoherent**.

```
import scalaz.syntax.semigroup._, scalaz.std.string._  
val h = Instance("hello")  
val w = Instance("world")
```

```
scala> h |+| w: String  
res2: String = worldhello
```

```
scala> (h: String) |+| w  
res3: String = helloworld
```

## Avoiding autowidening without proving equality

You can prove conformance (is subtype of) instead, in a very subst-like way.

```
def substCo[F[+_]](f1: F[Label]): F[String]

// still in the signature
def substContra[F[-_]](fs: F[String]): F[Label] = {
  type K[+A] = F[A] => F[Label]
  substCo[K](identity)(fs)
}
```

This hints that variance is at the *heart* of subtyping.

## Let's avoid subtyping instead

Allows only one-way conversion with maximum  $O(1)$  support, and no autowidening or variance declarations required.

```
type Label // still in the signature
```

```
def substCo[F[_]: Functor](f1: F[Label]): F[String] = {
  type K[A] = F[A] => F[String]
  implicit val K = Contravariant[? => F[String]].icompose[F]
  substContra[K](identity).apply(f1)
}
```

```
def substContra[F[_]: Contravariant](fs: F[String])
  : F[Label]
```

# Richer constraints for new features

Validating traversal without reallocating:

```
def validate[F[_]: Foldable](fs: F[String])
    : Option[F[Label]] =
  if (fs.all(_ == "42")) Some(fs) else None
```



# A list is an infinite tower of optional tuples

```
// Option[(A, Option[(A, Option[(A, ...)])])]  
type XList[+A] = Option[(A, XList[A])]  
// illegal cyclic reference ^ involving type XList  
  
final case class XList[+A](uncons: Option[(A, XList[A])])
```

# It's really a list

```
def fromList[A](xs: List[A]): Instance.XList[A] =
  xs.foldRight(Instance[A](None)){
    (x, xs) => Instance(Some((x, xs)))
  }
```

```
scala> fromList(List(1, 2, 3))
res0: xlist.XListModule.Instance.XList[Int]
     = Some((1, Some((2, Some((3, None))))))
```

Scalaz 8 List is defined this way, more or less:

<https://github.com/scalaz/scalaz/pull/1455>

# The recursion makes this tricky

- XList **must** appear in its own definition to be correct
- But, **no** code can see this self-recursion happening
  - not the *signature*
  - not the *binding to val*
  - not even the *structure!*

## The solution: use the val

```
sealed abstract class XListModule {
  type XList[+A]
  def apply[A](one: Option[(A, Instance.XList[A])]): XList[A]
  def uncons[A](xl: XList[A]): Option[(A, Instance.XList[A])]
}
```

```
val Instance: XListModule = new XListModule {
  type XList[+A] = Option[(A, Instance.XList[A])]
  def apply[A](one: XList[A]): XList[A] = one
  def uncons[A](xl: XList[A]): XList[A] = xl
}
```

# Completing the circle

- 1 structure implements signature
- 2 val is initialized by structure
- 3 signature uses val in declarations

# The flow of types into Instance

```
val Instance: XListModule = new XListModule {
// 3      2                1
```

- 1 Structure is created; knows RHS of XList, but **not** that its XList will become Instance.XList
- 2 Ascription : XListModule *existentializes* XList, hiding RHS
- 3 Instance is set as the *stable path* to the existential XList, forming the global existential type Instance.XList

# Can you do this in Java?

Sure, just put your whole program inside a single generic method.

# Primitives and boxing

This can be interesting, but programmers who hear about it fall in the tarpit and forget the main reason for thinking about this in the first place, *improving abstraction*.



## Intersection-style newtypes

You can write a no-cast structure for this signature.

```
type Extra[T]
```

```
def addExtra[A, T](a: A): A with Extra[T]
```

# SIP-35 Opaque types

- Cool, but. . .
- This works today (and has for several years)
- No way to do most of what I've shown in this talk
  - Same goes for newtype macro libraries
  - Same goes for type tags in libraries

## References

- This presentation source and example code, <https://launchpad.net/opaque-types-to-infinity>
- Julian Michael's original email, <https://is.gd/LowerCaseString>
- Article on related techniques, "...and the glorious subst to come", <https://is.gd/GloriousSubst>
- "Liskov Substitution Principle is Contravariance", <https://is.gd/Z48ext>
- Set scalac option `-Xsource:2.13` if having trouble with implicits: <https://github.com/scala/scala/pull/6074>

Copyright ©2018 Stephen Compall. This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>