# Lightweight Higher-kinded Dependent Object Types

CAMERON WONG, Harvard University, USA

Scala, like many other functional programming languages, has first-class support for higher-kinded type abstraction. However, its core calculus Dependent Object Types (DOT) does not, complicating the design and implementation of the feature in modern Scala compilers.

We demonstrate a novel encoding of higher-kinded types into Scala using only features known to be expressible in DOT via defunctionalization.

Additional Key Words and Phrases: type constructor polymorphism, higher-kinded types, higher-order genericity, Scala

## 1 INTRODUCTION

Higher-kinded types enable a greater level of abstraction by allowing code to be generic over type *constructors* rather than fully-realized types. For example, a programmer may design a data structure that abstracts over a container type to enable variants of the same structure to share the same definition.

Like many other functional languages, Scala supports higher-kinded types. However, the DOT calculus (DOT) [Amin et al. 2016] underlying the Scala 3 Dotty compiler does not. This means that there is no formal model of how higher-kinded types interact with Scala's other features in full generality, making it difficult to reason about the correctness of their implementation in Dotty, which uses an extension to DOT that has not been proven sound [Odersky et al. 2016].

What is so difficult about extending DOT's soundness proof to include higher-kinded generics? While a full exploration of DOT and its metatheory is beyond the scope of this paper, one reason is that DOT does not actually have primitive generics at all. Instead, a parameterized type like `List[T]` is represented with type members:

```
class List:
  type T
  // further definitions, methods, etc.
```

with specific instantiations of the parameter `T` being provided via specialization:

```
class ListInt extends List:
  type T = Int
  // etc.
```

In this way, abstract type members can be used as a form of first-order polymorphism, where expressions and types can refer to type variables of kind $*$, the kind of regular types.

Our contribution is to demonstrate an idiomatic encoding of higher-kinded polymorphism using only such first-order features, in the hopes that this may direct further efforts to bridge the gap between Scala and DOT.

## 2 HIGHER-KINDED TYPES

We motivate higher-kinded types with two examples.

*Higher-kinded Data.* This example was first presented by [Macguire 2018]. Consider a datatype of `Person`, containing a name and an age. Objects in our program may be populated by a source, such as a webform, in which individual fields may be malformed without invalidating the data structure as a whole. However, within the program, we would like to ensure that all Persons have all valid fields. A naive modeling might use two separate classes (figure 1).

```
class Person:
  var name: String
  var age: Int

class UnvalidatedPerson:
  var name: Option[String]
  var age: Option[Int]
```

Fig. 1. Two datatypes, related only in name

This is quite an unsatisfying pair of definitions. The types Person and UnvalidatedPerson are only nominally related, meaning that if a new field is added, *both* types need to be updated, creating a maintenance burden. Furthermore, although the relationship is obvious, any updates will need to be done manually, without compiler assistance, introducing the potential for human error.

Both Person and UnvalidatedPerson have the same field names and "moral" types, differing only in the "shape" of said field type (T versus Option[T]). This suggests that, if we could abstract over that "shape", we could allow Person and UnvalidatedPerson to share a skeleton PersonImpl, allowing the compiler to enforce the relationship between them.

```
class PersonImpl[F[_]]:
  var name: F[String]
  var age: F[Int]

UnvalidatedPerson = Person[Option]
Person = Person[Id]
```

Fig. 2. Unifying two nominally-related datatypes via higher-kinded types

Higher-kinded types provide exactly this mechanism, in which PersonImpl takes a type-level function F and applies it to its two fields to produce the final data structure.

*Higher-ranked typeclasses.* Our second example is the venerable monad interface. A type constructor M is a monad if it admits the following functions:

```
def pure[A](x: A): M[A]
def bind[A, B](x: M[A], f: A => M[B]): M[B]
```

following the monad laws.

How can we *abstract* over all possible monads? A naive attempt at encoding the above functions as a trait might look as follows:

```
trait Monad[A]:
  def bind[B](f: A => /* ??? */): /* ??? */
def map[A, B](f: A => B, l: List[A]) : List[B] =
  l match
    case Nil => Nil
    case Cons(x, xs) => Cons(f(x), map(f, xs))

val foo = map(((x: Int) => x+1), nums)
val bar = map(((y: Bool) => !y), bools)
```

Fig. 3. A simple program using higher-order values

However, an issue arises when attempting to represent the type `M[B]`, as we have no way to refer to the underlying type constructor `M`, only the fully-applied `M[A]` implicitly as `self`.

With higher-kinded type members, however, we can solve this directly:

```
trait MonadOps:
  type M[_]
  def pure[A](x: A): M[A]
  def bind[A, B](x: M[A], f: A => M[B]): M[B]
```

Note that this trait must be instantiated on a separate object than the type `M` itself, and that object must be provided as an additional argument to any functions that are monad-agnostic. For example,

```
def select[A](
    ops: MonadOps,
    mb: ops.M[Boolean],
    left: ops.M[A],
    right: ops.M[A]): ops.M[Unit] =
  ops.bind(mb, (b : Boolean) => if b then left else right)
```

Of particular interest is that the function's type signature is able to refer to the type member `M` of the argument `ops`. The ability to use ad-hoc polymorphism via explicit dictionary-passing in this way is a major strength of Scala and path-dependent types.

## 3 DEFUNCTIONALIZATION

The problem of abstracting over higher-kinded type expressions in a language with only first-order type variables is well-known. Yallop [Yallop and White 2014] demonstrated how *defunctionalization* at the type level could be used to represent higher-kinded programs in OCaml.

Defunctionalization was initially introduced by Reynolds as a technique to express higher-order *functions* in a first-order language [Reynolds 1972]. To perform this transformation, we replace all occurrences of first-class functions with regular values, which can be eliminated with a special `apply` function. Consider a short program using first-class functions (figure 3).

To defunctionalize this program, we define a data type with a variant for each function value used, in this case once for `foo` and once for `bar`. Then, applications are replaced with calls to a specialized `apply` function (figure 4).

The argument to `map` is now an ordinary, non-function object, meaning the program as a whole requires no higher-order values[1]!

[1]Classically, the `Fun` trait would be a variant type, with the `apply` function dispatching via pattern matching instead of inheritance.

```
sealed trait Fun[A,B]:
  def apply(input: A): B

object Succ extends Fun[Int, Int]:
  def apply(x: Int): Int = x+1

object Neg extends Fun[Bool, Bool]:
  def apply(y: Bool): Bool = !y

def map[A, B](f: Fun[A,B], l: List[A]) : List[B] =
  l match
    case Nil => Nil
    case Cons(x, xs) => Cons(f.apply(x), map(f, xs))

val foo = map(Succ, nums)
val bar = map(Neg, bools)
```

Fig. 4. A defunctionalized program

### 3.1 higher in OCaml

At the type level, things work similarly. Yallop wrote a library, which he named `higher`, for higher-kinded programming in OCaml. As we are re-presenting his work, all code samples in this section will also be in OCaml.

We introduce an abstract type constructor

```
type ('a, 'f) app
```

where the type expression `(s, t) app` represents the application of the type expression `t` to the type expression `s` [2]. Importantly, the expression `t` here is a proper type – that is, of kind $*$. Type constructors are associated with an uninhabited phantom type, which Yallop called the *brand*. Beta-reduction does not occur automatically but is performed explicitly via introduction and eliminator functions for each brand. For example, the constructor *list* would have a corresponding module containing the brand *List.t* and projection and injection functions:

```
module List : sig
  type t
  val inj : 'a list -> ('a, t) app
  val prj : ('a, t) app -> 'a list
end
```

Now, abstraction over type constructors can be expressed by abstracting over brands, e.g.

```
type 'f monad =
  { pure : ∀'a . 'a -> ('a, 'f) app
  ; bind : ∀'a 'b . ('a, 'f) app -> ('a -> ('b, 'f) app) -> ('b, 'f) app
  }
```

[2]OCaml type expressions are written "backwards" – in the type `t list`, the argument `t` comes first. Yallop chose to use the same convention in the ordering of the arguments to app.

One concern with defunctionalization is that it is classically a whole-program transformation – the single sum type app represents *all* higher-order type constructors in the program. As an intermediate compiler representation, this is no obstacle, but higher is intended for use as an end-user library, and it would not be ergonomic to require that the programmer know all higher-kinded occurrences up front.

How is the mysterious type app defined? Yallop provided two implementations and alluded to a third, using an unsafe cast, an extensible variant type and type families respectively. We detail only the extensible variant approach, as it most closely resembles the equivalent Scala construction.

```
type ('a, 'f) app = ..

module Newtype(T : sig type 'a t end)() = struct
  type 'a s = 'a T.t
  type t
  type (_, _) app += App : 'a s -> ('a, t) app
  let inj v = App v
  let prj (App v) = v
end
```

Fig. 5. Implementing higher with open variants

Figure 5 shows this implementation of higher. The first line declares that app is an open data type whose constructors will be declared later. Next is the Newtype functor (a module-level function) which takes, as input, a module declaring a generic type 'a t, which extends the app type with a new variant App mapping 'a s to ('a, t) app. The prj and inj functions then use App. Note that the match in the prj function is technically inexhaustive, relying on the fact that t is fresh per invocation of Newtype to guarantee a match in practice[3].

## 3.2 From OCaml to Scala

In this section, we demonstrate how a similar transformation can be applied in surface-level Scala. Unlike OCaml, Scala natively supports higher-kinded polymorphism, so we must ensure that we do not cheat by trivially using the thing we seek to encode! Specifically, we must ensure that all instances of polymorphic types *and* abstract type members occur at kind $*$. Other forms of first-order polymorphism (namely, polymorphic classes and traits) are allowed, as these can be expressed in DOT.

The most important part of this translation is the trait Apply[F,A], corresponding to the OCaml type ('a, 'f) app. In OCaml, we used generalized algebraic datatypes to associate the underlying 'a T.t with the defunctionalized ('a, t) app, using pattern matching to define projection and injection. In Scala, we can use type members for this purpose:

```
trait Apply[F,A]:
  type This
  def prj(): This
```

This definition has a few key differences from the OCaml implementation. Instead of encapsulating the type in a module, we attach the projection function to the object itself. Also of note is

[3]It is also important to disallow any downstream clients from extending app via OCaml's type privacy controls; otherwise a user may manually define a new constructor with the same type as App.

the the abstract type member `This`, which can refer to the bound parameter `A` and represents the fully-realized type. Finally, if the trait is not sealed, we are automatically extensible, allowing us to easily add new type constructors and their brands. This is a double-edged sword, as we will see in section 4.

Instantiations of this trait for `List` and `Option` are shown in figure 6.

```
object ListW:
  case object Brand
  type T = Brand.type

  class Inj[A](val x: List[A]) extends
      Apply[T, A]:
    type This = List[A]
    override def prj(): This = x
```

```
object OptionW:
  case object Brand
  type T = Brand.type

  class Inj[A](val x: Option[A])
      extends Apply[T, A]:
    type This = Option[A]
    override def prj(): This = x
```

Fig. 6. Two instantiations of `Apply`.

Although the inner class `Inj` takes a parameter, it is allowed because it is not abstract; the nesting within the outer `object` is merely for scoping purposes.

Unfortunately, although these implementations are formulaic, they must be written by hand. The input to the `Newtype` functor contains a type member `'a t`, which is *exactly* the feature (in Scala) we are encoding in the first place[4]! This will be discussed further in section 4.

## 3.3 Examples

We can now easily represent higher-kinded data by abstracting over the brand:

```
class PersonImpl[F]:
  var name: Apply[F, String]
  var age: Apply[F, Int]

type UnvalidatedPerson = PersonImpl[OptionW.T]
type Person = PersonImpl[IdW.T]
```

Defining typeclass constraints, like the monad interface, can be done by further extending `Apply`:

```
trait Monad[F, A] extends Apply[F, A]:
  def pure(x: A): Monad[F, A]
  def bind[B](f: A => Monad[F, B]): Monad[F, B]

def select[M, A](
    mb: Apply[M, Boolean],
    left: Apply[M, A],
    right: Apply[M, A]): Apply[M, Unit] =
  mb.bind((b : Boolean) => if b then left else right)
```

Notice that `select` no longer needs to take the explicit ops argument, allowing the more idiomatic form `mb.bind`.

[4]Unlike Scala, the OCaml module language is separate from its type and expression languages. While OCaml does allow modules to be used as term-level values, they are far more restricted than Scala's objects.

## 4 CONCLUSION

We have demonstrated that type-level defunctionalization can be used to represent higher-kinded polymorphism in Scala while remaining within the surface area of features expressible in DOT. In this section, we discuss the potential benefits, alongside further development directions.

### 4.1 Limitations in handwritten code

Compared to `higher`, type-level defunctionalization in Scala has a number of downsides making it unsuitable for use by the end-user. This is of no immediate concern, as Scala supports higher-kinded polymorphism natively (so end-users have no need to rely on this pattern to express such programs in the first place), but we outline their limitations below for completeness.

*4.1.1 Brand uniqueness.* The safety of this transformation relies on there being a unique instantiation of `Apply[F, A]` for any given brand `F` and argument `A`. However, this is not guaranteed. Consider a violating consumer manually associating `Apply[ListW.T, A]` with `Int` (figure 7).

```
object ListW:
  case object Brand
  type T = Brand.type
  // ... (see fig. 6)

class Bad[A] extends Apply[ListW.T, A]:
  type This = Int
  override def prj(): This = 0
```

Fig. 7. A careless end-user may break the guarantees of `Apply`

In OCaml, the generativity of functors allowed the definition of `app` to be hidden from outside consumers, preventing downstream code from generating new variants other than as allowed by the `Newtype` functor. Unfortunately, there is no obvious way to recreate this in Scala with sealed traits, as we cannot specify the type of the input to a theoretical `newtype` function without the use of abstract, higher-kinded type members.

*4.1.2 Kind checking.* Similarly, the defunctionalized encoding contains no built-in kind checking of any sort, limiting the feedback a programmer will receive on ill-kinded uses. While the trait system can be used to prevent instantiations of ill-kinded applications like `Apply[ListW.T, ListW.T]` by ascribing brands to generated traits such as, e.g. `TtoT`, this is cumbersome and error-prone.

There is also nothing stopping end-users from creating a value of type `ListW.T` directly, either by referring to `ListW.Brand` or via a typed `null`. We believe this to be a minor drawback, as it requires intentional action by the progrmamer that is difficult to perform by accident.

### 4.2 As a compiler representation

On the other hand, neither issue is a concern if defunctionalization is performed completely automatically. By using internal, mangled names, a programmer can never refer to either the trait `Apply` or witness objects such as `ListW` to manually create erroneous instantiations. Furthermore, the boilerplate of the witness objects lend themselves well to automation.

Similarly, kind checking can be delegated to the surface language before being erased in a defunctionalization pass. Ill-kinded snippets such as `List[List]` or `x : List` can be ruled out during elaboration, ensuring that only well-kinded applications remain.

## 4.3 Related and future work

*4.3.1 Implementation details.* This implementation of type-level defunctionalization leans heavily on the existence of first-order parametric types in the target language. In theory, this is of little concern, as Amin et al demonstrated that these can be easily expressed in DOT [Amin 2016]. In practice, there are still many design questions left unanswered. The simple, DOT-inspired representation of generics in dotty was ruled unviable due to expressivity concerns [Odersky et al. 2016], and the problematic cases overlap with the construction presented in this paper. Instead, a defunctionalization pass might be used to simplify the surface language to a point that new encoding is revealed.

*4.3.2 Relating to DOT.* Stucki et al [Stucki 2017] presented an extension to System $F_\omega$ featuring interval kinds that enforced sub- and supertype bounds at the kind level, which was sufficient to express many of Scala's patterns but were unable to extend their results to support full path-dependence.

One goal of this work is to argue that DOT can express higher-kinded patterns as-is. However, the translation presented here uses several high-level features that are not present in DOT, such as traits and rank-1 generics. To rule out a higher-kinded extension to DOT entirely, a logical next step would be to show a translation from a higher-kinded extension to DOT to DOT itself. As a first step, we outline a possible approach in appendix A.

*4.3.3 Simplification.* As presented in this paper, instantiations of the `Apply` trait requires the creation of a separate class `Inj[A]` to wrap the underlying type. This is because, unlike Haskell's typeclasses or Rust traits, Scala's traits cannot be applied to a type post-hoc. If we allow the compiler to insert further traits to a class at definition time, a far more direct implementation becomes possible (figure 8).

```
case object ListW // Compiler-generated, inaccessible in the surface language
class List[A] extends Apply[ListW, A]:
  // ...
```

Fig. 8. A simpler way to instantiate `Apply`

By attaching the `Apply` trait to the underlying class in question, we obviate the need for explicit projection and injection functions entirely.

This simpler encoding has two major downsides. Firstly, type aliases such as `type Foo[A] = String` have no underlying class to which to attach the trait. Secondly, this encoding admits no obvious way to express a partially-applied or curried type constructor. If these can be resolved (or sufficiently worked-around), this may prove to be a superior first-order representation.

## ACKNOWLEDGMENTS

Thanks to Prof. Amin for graciously allowing me to change my final project after my initial proposal didn't work.

## REFERENCES

Nada Amin. 2016. *Dependent Object Types.* PhD thesis. EPFL. https://infoscience.epfl.ch/record/223518 (code).

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. *The essence of dependent object types.* Lecture notes in computer science, Vol. 9600. Springer International Publishing, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Karl Crary. 2019. Fully Abstract Module Compilation. *Proc. ACM Program. Lang.* 3, POPL, Article 10 (jan 2019), 29 pages. https://doi.org/10.1145/3290323

Sandy Macguire. 2018. Higher-Kinded Data. Blog article.

Martin Odersky, Guillaume Martres, and Dmitry Petrashko. 2016. *Implementing higher-kinded types in Dotty.* ACM Press, 51–60. https://doi.org/10.1145/2998392.2998400

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

Sandro Stucki. 2017. Higher-Order Subtyping with Type Intervals.

Jeremy Yallop and Leo White. 2014. *Lightweight Higher-Kinded Polymorphism.* Lecture notes in computer science, Vol. 8475. Springer International Publishing, 119–135. https://doi.org/10.1007/978-3-319-07151-0_8

# Appendices

## A DEFUNCTIONALIZATION IN DOT

In this section, we outline how type-level defunctionalization may be performed in DOT proper.

We leave the specifics of the source language intentionally vague, as its design will likely strongly inform the translation. To sufficiently model the surface language, it would likely have explicit type-level lambdas and associated kinding/subtyping rules.

Following Amin et al [Amin 2016], the translation of the `Apply` trait is straightforward:

$$\begin{aligned} \texttt{Apply[F, A]} \triangleq \mu(\texttt{self}.( & \quad \{F : *\} \\ \wedge & \quad \{A : *\} \\ \wedge & \quad \{T : \bot \cdot\cdot \{A : *\}\} \\ \wedge & \quad \{\texttt{prj} : \texttt{self}.T\})) \end{aligned} \tag{1}$$

The translation of types would most likely be kind-directed, rather than purely syntactic. This is because a type constructor of arrow kind $k_1 \rightarrow k_2$ must generate a brand and instantiation of `Apply` that can be referred to by the rest of the program, but proper types can be translated more straightforwardly. Following Crary [Crary 2019], we might define the judgment $\Gamma \vdash T : k \rightsquigarrow [A.\overline{T}, \overline{T}]$ to separate witness carriers (which need to be lifted to the top level) from the final translation (which is used in-place). We bind the variable $A$, representing the parameter to the lambda, which will eventually be re-bound when instantiating the `Apply` trait at the top level.

A candidate translation is suggested in figure 9.

$$\frac{\Gamma, A : k \vdash T : k' \rightsquigarrow [A'.\overline{T'}, \overline{T}] \qquad \mathsf{fresh}(\mathsf{brand})}{\Gamma \vdash \lambda(A : k).k' : k \rightarrow k' \rightsquigarrow [\_.\mu(\texttt{self}.\{A : *\} \wedge \overline{T} \wedge \overline{T'}[\texttt{self}.A/A']), \mathsf{brand}]}$$

Fig. 9. Sample translation rule for type lambdas

As written, this rule is most likely incorrect; it does not correctly handle scoping for nested lambdas, which is crucial for representing curried multi-parameter type constructors. Even so, it is hopefully illustrative of what the final translation rules may look like.

A secondary concern is that the correctness of defunctionalization relies on the ability to generate infinitely-many fresh brands. While DOT lacks direct nominal typing, it can be faked by using fresh label names, which will never unify with a pre-existing type in the program.