Implementing Higher-Kinded Types in Dotty

Martin Odersky, Guillaume Martres, Dmitry Petrashko

EPFL, Switerland: {first.last}@epfl.ch

Abstract

dotty is a new, experimental Scala compiler based on DOT, the calculus of Dependent Object Types. Higher-kinded types are a natural extension of first-order lambda calculus, and have been a core construct of Haskell and Scala. As long as such types are just partial applications of generic classes, they can be given a meaning in DOT relatively straightforwardly. But general lambdas on the type level require extensions of the DOT calculus to be expressible. This paper is an experience report where we describe and discuss four implementation strategies that we have tried out in the last three years. Each strategy was fully implemented in the *dotty* compiler. We discuss the usability and expressive power of each scheme, and give some indications about the amount of implementation difficulties encountered.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Polymorphism

General Terms Languages, Compilers, Experimentation

Keywords type constructor polymorphism, higher-kinded types, higher-order genericity, Scala, dotty, DOT, dependent object types

1. Introduction

Scala has first-class support for higher-kinded types [3], they can be defined by users as follows:

```
type Foo[A] = List[A] // Foo has kind * -> *
```

and abstracted over:

```
def return[F[_], A](x: A): M[A]
type Bar[M[_]] = M[Int] // Bar has kind (* -> *) -> *
```

Implementing sound support for these higher-kinded types in *dotty* [5] without restricting their expressive power proved to be challenging, so much so that we evaluated four different strategies before settling on the current direct representation encoding. The strategies are summarized as follows:

- A *simple encoding* in the DOT-inspired [9] core type structures that can express partial applications and not much more
- A *direct representation* that adds support for full type lambdas and higher-kinded applications, without reusing much of the existing concepts of the calculus and the compiler.
- A *projection encoding*, that encodes higher-kinded types as first-order generic types using type projections `T#A`.
- A *refinement encoding*, that encodes higher-kinded types as first-order generic types using refinements and path-dependent types.

Neither of the encodings is fully transparent, in that some type checking operations still needed special provisions for encoded types.

These four strategies were implemented in the *dotty* research compiler for Scala over the course of three years (2013-2016). The purpose of the present paper is to give a high-level overview of the implementations and the language design choices they entail.

The perspective of the paper is experimental rather than theoretical. One can regard it as a kind of lab notebook describing and contrasting different experiments. The raw data for the experiments exists in the form of commits in the repository 'lampepfl/dotty' on GitHub. Given the considerable implementation effort that went into higher-kinded types we wanted to create a record of what was done, what worked out, and what did not work as well as hoped for. Overall, it's fair to say that there were more failed than successful experiments, but failures are at least as important to record as successes.

The rest of this paper is organized as follows. Section 3 describes the simple encoding of partial applications into core DOT. Section 4 describes the direct representation of higher kinded types. Section 5 and Section 6 describe two encodings based on projections and refinements, respec-

SCALA'16, October 30–31, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4648-1/16/10...
http://dx.doi.org/10.1145/2998392.2998400

tively. Section 7 compares the four implementation strategies described previously. Section 8 concludes.

2. Background

If we combine generics and subtyping in a language like Java or Scala, we face the problem that we want to express a generic type where the type argument is an *unknown* type that can range over a *set of possible types*. The prototypical case is where the argument ranges over all subtypes or supertypes of some type bound, as in `List[_ <: Fruit]`.

Such partially undetermined types come up when we want to express variance. We would like to express, say, that `List[Apple]` is a subtype of `List[Fruit]` since `Apple` is a subtype of `Fruit`. An equivalent way to express this is to say that the type `List[Fruit]` includes `Lists` where the elements are of an arbitrary subtype of `Fruit`. By that reasoning, `List[Apple]` is a special case of `List[Fruit]`. We can also express this notion directly using the wildcard type `List[_ <: Fruit]`. *Definition-site variance* can be regarded as a user-friendly notation that expands into *use-site variance* expressions using such wildcards.

The problem is how to model a wildcard type such as `List[_ <: Fruit]`. Igarashi and Viroli's original interpretation [1] was as an existential type $\exists$`T <: Fruit.List[T]` which would be written

```
List[T] forSome { type T <: Fruit }
```

in current Scala. However, existential types usually come with explicit pack and unpack constructs [2], which are absent in Scala's setting. Moreover, actual subtyping rules as e.g. implemented in the reference compilers for Java and Scala are more powerful than what can be expressed with existential types alone [12]. The theory of the rules that are actually implemented is not fully known and the issues look complicated. Tate, Leung and Learner have explored some possible explanations in [11], but their treatment raises about as many questions as it answers.

2.1 A Uniform Representation of Types

The problem is solved in DOT and *dotty* by a radical reduction. Type parameters and type arguments are not primitive, but are seen as syntactic sugar for type members and type refinements. For instance, if `List` is declared like this:

```
trait List[Elem] { ... }
```

then this would be expanded to a parameterless trait with a type member, like this:

```
trait List { type Elem; ... }
```

(For simplicity we re-use the name of the parameter `Elem` as the name of the type member, whereas in practice the compiler would choose a mangled name like `List$Elem` in order to avoid name clashes.)

An application such as `List[String]` is then expanded to `List { type Elem = String }`. If `List` were declared as covariant (using `[+Elem]`), the type application is instead expanded to a refinement with an upper bound:

```
List { type Elem <: String }
```

Analogously, applications of contravariant types lead to refinements with lower bounds.

This scheme has two immediate benefits. First, we only need to explain one concept instead of two. Second, the interaction between the two concepts, which was so difficult before, now becomes trivial. Indeed, a type like

```
List[_ <: Fruit]
```

is simply

```
List { type Elem <: Fruit }
```

That is, wildcard parameters translate directly to refinements with the same type bounds.

3. The Simple Encoding

Following DOT, we model type parameters as type members and type arguments as refinements. For instance, a parameterized class such as

```
Map[K, V]
```

is treated as equivalent to a type with type members:

```
class Map { type Map$K; type Map$V }
```

The type members are name-mangled (i.e. `Map$K`) to ensure that they do not conflict with other user-defined members or parameters named `K` or `V`.

A type-instance such as `Map[String, Int]` would then be treated as equivalent to

```
Map { type Map$K = String; type Map$V = Int }
```

whereas a wildcard type such as `Map[_, Int]` is equivalent to:

```
Map { type Map$V = Int }
```

That is, _ arguments correspond to type members that are left abstract. Wildcard arguments can have bounds. E.g.

```
Map[_ <: AnyRef, Int]
```

is equivalent to:

```
Map { type Map$K <: AnyRef; type Map$V = Int }
```

3.1 Type Parameters and Partial Applications

The notion of type parameters makes sense even for encoded types, which do not contain parameter lists in their syntax. Specifically, the type parameters of a type are a sequence of

type fields that correspond to parameters in the unencoded type. They are determined as follows.

- The type parameters of a class or trait type are those parameter fields declared in the class that are not yet instantiated, in the order they are given. Type parameter fields of parents are not considered.
- The type parameters of an abstract type are the type parameters of its upper bound.
- The type parameters of an alias type are the type parameters of its right hand side.
- The type parameters of every other type is the empty sequence.

This definition of type parameters leads to a simple model of partial applications. Consider for instance:

```
type Histogram = Map[_, Int]
```

`Histogram` is a higher-kinded type that still has one type parameter. `Histogram[String]` would be a possible type instance, and it would be equivalent to `Map[String, Int]`.

One interesting consequence of this definition is that higher-kinded types and existential types are identified with each other by virtue of being mapped to the same construct. Indeed, the type `Map[_, Int]` can be interpreted as both an existential type, where the `K` field is unspecified and as a higher-kinded type that takes a type argument for the `K` field and produces an instance of `Map`.

3.2 Modeling Polymorphic Type Declarations

The partial application scheme gives us a new – and quite elegant – way to express certain higher-kinded types. But how do we interpret the polymorphic types that exist in Scala?

More concretely, Scala allows us to write parameterized type definitions, abstract types, and type parameters. In the new scheme, only classes (and traits) can have parameters and these are treated as equivalent to type members. Type aliases and abstract types do not allow the definition of parameterized types so we have to interpret polymorphic type aliases and abstract types specially.

Parameterized Aliases. A simple, and quite common case of parameterized type definitions in Scala are parameterized aliases. For instance, we find in the Scala package the definition

```
type List[+T] =
   scala.collection.immutable.List[T]
```

Aliases like these can be expanded under the simple encoding by simply dropping the parameters on the left hand side and the arguments on the right hand side of the equals sign.

Partial Applications. Type definitions representing partial applications like `Histogram` above are straightforward.

Non-linear parameter occurrences. It is also possible to express some patterns where type parameters occur non-linearly on the right-hand side. An example is the definition of `Pair` below.

```
type Pair[T] = Tuple2[T, T]
```

where `Tuple2` is declared as

```
class Tuple2[+T1, +T2] ...
```

The definition of `Pair` is expanded to the following parameterless type alias:

```
type Pair =
  Tuple2 { type Tuple2$T2 = Tuple2$T1 }
```

More generally, each type parameter of the left-hand side must appear as a type member of the right hand side type. Type members must appear in the same order as their corresponding type parameters. References to the type parameter are then translated to references to the type member. The type member itself is left uninstantiated.

3.3 Limitations

The technique described in the previous section can expand most polymorphic type aliases appearing in Scala codebases but not all of them. Here are some examples of types that *cannot* be expressed:

1. `type Rep[T] = T`

   This fails because the right hand side `T` does not have a type field named `T`.

2. `type LL[Elem] = List[List[Elem]]`

   This fails because the occurrence of the parameter `Elem` on the right hand side is not a member binding of the outer `List`.

3. `type RMap[V, K] = Map[K, V]`

   This fails because the order of type parameters of the left- and right-hand sides of the definition differ.

Another restriction concerns the bounds of higher-kinded type parameters. Consider the following pattern:

```
class Seq[X] extends Iterable[X] ...

def f[C[X] <: Iterable[X]]: C[String] = ...
def g[C[X] <: Seq[X]]: C[String] = f[C]
```

According to our rules for type parameters, the result type of `f` is encoded as

```
C { type Iterable$X = String }
```

On the other hand, the result type of `g` is encoded as

```
C { type Seq$X = String }
```

The two types are incompatible, hence the example above would be lead to an ill-typed encoding, even though it seems completely natural. The problem here is that type parameters are encoded as type fields with mangled names that contain the name of the enclosing class. This means that narrowing of bounds for type parameters is not supported. The root problem in the example above is that the type parameter `C` in `g` has a type parameter field named `Seq$X` whereas the type parameter `C` in `f` has a type parameter named `Iterable$X`. Therefore, it should not be allowed to pass `C` from `f` to `g`.

In a sense the simple encoding abandons the traditional notion of kinds, but replaces it with the notion that the kind of a type is the sequence of the names of its type parameter fields. According to the new notion, the call `f[C]` above would not be kind-correct.

This discussion also points to a need for a mechanism to enforce that the type parameters of a class have the same names as the type parameters of a superclass. In the example above, we would like to enforce that the type parameter of class `Seq` has the same (encoded) name as the type parameter of class `Iterable`. A possible way to do this would be by allowing explicitly named parameters that are available under the same name as public fields. E.g.,

```
type Seq[type X] extends Iterable[X]
```

A more detailed discussion of named type parameters is beyond the scope of this paper.

3.4 Discussion

The simple encoding has the advantage that no new concepts beyond those already covered by DOT are needed. It supports all forms of partial application naturally, with minimal notational overhead.

On the other hand, the limitations of the simple encoding make it less expressive than the current implementation of higher-kinded types in Scala. Furthermore, the distinction between what can be expressed and what cannot looks somewhat arbitrary and not well connected with the source-level parameter syntax.

4. The Direct Representation

The direct representation of higher-kinded types keeps the encoding of type parameters of traits and classes in terms of type members as before. Higher-kinded abstractions and applications are modeled by their own constructs. In particular, we add explicit internal representations for:

- Type lambdas

  `[`v_1 `X`$_1$`, ...,` v_n `X`$_n$`] -> T`

  where $v_1, \ldots, v_n$ are the variances of the type parameters. This is used internally but can also be written explicitly by the user (see Section 4.2), in fact

  ```
  type Foo[+X] = T
  ```

  is now just syntactic sugar for:

  ```
  type Foo = [+X] -> T
  ```

- Higher-kinded applications

  `C[`T_1`, ...,` T_n`]`

  where `C` is a higher-kinded type constructor and T_1, ..., T_n are argument types. In such an application `C` is always a higher-kinded abstract or alias type or a type parameter. If `C` is a class, the usual encoding with refinement types is applied. If `C` is a lambda abstraction, beta reduction is applied:

  ```
  ([v1 X1, ..., vn Xn] -> T)[U1, ..., Un]
  -->
  [X1 := U1, ..., Xn := Un]T
  ```

  Reducing applied aliases proceeds similarly, but this is not done eagerly in general as it affects type inference, see the example at the end of this section.

We now sketch the extra subtyping rules as they are implemented in the compiler (a more formal treatment would require us to extend the soudness proof of DOT), for simplicity of presentation we only cover the case of single-parameter lambdas and we do not consider F-bounds or kind-checking. The rule governing type lambdas is as follows (*conforms*(v_1, v_2) specifies the conformance relation between variances. It is **true** iff $v_1 = v_2$ or if v_2 is non-variant) :

$$\frac{\begin{array}{c}\mathit{conforms}(v_1, v_2)\\ \Gamma \vdash L_1 <: L_2\\ \Gamma \vdash U_2 <: U_1\\ \Gamma, X >: L_1 <: U_1 \vdash T_1 <: T_2\end{array}}{\Gamma \vdash [v_1 X >: L_1 <: U_1] \to T_1 <: [v_2 X >: L_2 <: U_2] \to T_2}$$

Subtyping rules for higher-kinded applications are as follows (here, the syntax $\Gamma \vdash S <:_! T$ or $\Gamma \vdash S >:_! T$ means that the closest known upper (respectively, lower) bound of type S is T).

$$\frac{\begin{array}{c}\Gamma \vdash A_1 <:_! [v\, X] \to U_1\\ \Gamma \vdash [X := T_1]U_1 <: U_2\end{array}}{\Gamma \vdash A_1[T_1] <: U_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_2 >:_! [v\, X] \to U_2\\ \Gamma \vdash U_1 <: [X := T_2]U_2\end{array}}{\Gamma \vdash U_1 <: A_2[T_2]}$$

$$\frac{\begin{array}{c}\Gamma \vdash A <:_! [+X] \to U\\ \Gamma \vdash T_1 <: T_2\end{array}}{\Gamma \vdash A[T_1] <: A[T_2]}$$

$$\frac{\Gamma \vdash A <:_! [-X] \to U \qquad \Gamma \vdash T_2 <: T_1}{\Gamma \vdash A[T_1] <: A[T_2]}$$

$$\frac{\Gamma \vdash A <:_! [X] \to U_1 \qquad \Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_1}{\Gamma \vdash A[T_1] <: A[T_2]}$$

Type inference also has to be adapted to higher-kinded types. The main addition needed concerns the case where the compiler needs to satisfy a subtyping constraint

```
S <: C[T1, ..., Tn]
```

where `C` is an instantiable higher-kinded type parameter and `S` and $\texttt{T}_1$, ..., $\texttt{T}_n$ are types. We need to find an instantiation of `C` that satisfies the constraint.

The scheme to find this instantiation is essentially the same as for the most recent version of *scalac* (including support for partial unification of type constructors [10]). We first find a base type of `S` that has a constructor with at least n type parameters where the variances of the rightmost n parameters conform to those of `C`'s type parameters. Let that base type be

```
B[S1, ... Sm, U1, ..., Un].
```

Then, try to instantiate `C` to

```
[X1, ..., Xn] -> B[S1, ... Sm, X1, ..., Xn]
```

and, if this succeeds, continue with the subtyping check

```
B[S1, ... Sm, U1, ..., Un] <:
B[S1, ... Sm, T1, ..., Tn]
```

The search for suitable base types proceeds according to the linearization order of `C`. This is a deviation from *scalac*, which uses a slightly different order in which base types are visited. In both compilers, once a base type satisfies both the type parameter instantiation and the subtyping check, the type parameter stays instantiated to that base type, even if subsequent subtyping checks fail. This is analogous to Prolog's "cut" operator that prevents backtracking from undoing a partial success. The cut is necessary to prevent a combinatorial explosion by limiting the search space.

Example: Assume the following definitions:

```
trait A[X]
trait B[X, Y]
object O extends A[String]
     with B[Int, String]
def f[C[X], Z](x: C[Z]) : C[Z] = x
```

Then the type parameters for `f` in the call `f(O)` are inferred as follows.

1. The constraint to be satisfied is `O <: C[Z]`, where `C` and `Z` are instantiable type variables.
2. The argument `O` does not have the right number of type parameters to match the pattern `C[Z]` and is therefore discarded.
3. The next base type in linearization order from `O` is `B[Int, String]`. This type has enough type parameters, and we therefore instantiate `C` to `[X] -> B[Int, X]`.
4. After instantiation we obtain the constraint

   ```
   B[Int, String] <: B[Int, Z],
   ```

   which leads to the instantiation of `Z = String`.
5. The call is hence expanded to

   ```
   f[[X] -> B[Int, X], String](O)
   ```

   and its result type is `B[Int, String]`.
6. One could have alternatively chosen to instantiate `C` to `[X] -> A[X]`. But since `B` came first in linearization order, this alternative was discarded. If subsequently we were faced with the constraint that the result type of `f` should be a subtype of `A[String]` this constraint will fail.

Inference takes abstract types and type aliases into account when trying to find possible type parameter instances. For instance, given the type alias

```
type Transform[X] = Map[X, X]
```

and the definition

```
val trans: Transform[String]
```

the call `f(trans)` would be expanded to

```
f[Transform, String](trans)
```

That is, `Transform` is a valid candidate for the decomposition of the type of `t` into a type constructor and a type argument. For this to work, it is important that aliases are not dereferenced eagerly in the compiler. If the compiler had expanded the binding

```
trans: Transform[String]
```

to

```
trans: Map[String, String]
```

type inference would have yielded a different, and less intuitive expansion:

```
f[[X] -> Map[String, X], String](trans)
```

4.1 Higher-Kinded Wildcard Applications

Recall that one of the main motivations of *dotty*'s encoding of type parameters was to give a simple semantics to wildcard arguments. With the introduction of higher-kinded applications, the problem resurfaces. For example, consider the definition:

```
type M[X] <: Map[X, X]
```

What should be the meaning of `M[_]` be? One might be tempted to simply disallow higher-kinded applications to wildcard arguments. But unfortunately, Scala libraries do contain occurrences of such applications, which are hard to work around. Another possible interpretation would be as an existential type - i.e. `M[_]` corresponds to

```
Map[X, X] forSome { type X }.
```

If we follow that line, every existential type in Scala could be expressed as a higher-kinded application to wildcard arguments. Indeed,

```
T forSome { type X >: L <: H }
```

is equivalently expressed as

```
([X] -> T)(_ >: L <: H).
```

On the other hand, getting rid of existential types was another design objective of the *dotty* project. In the absence of explicit pack and unpack constructs, their interactions with many other concepts are unclear. Furthermore, existential types are semantically quite close to path-dependent types and it seems undesirable to have two concepts that largely overlap.

The solution pursued in *dotty* is to disallow applications of higher-kinded types to wildcards unless these applications can be ultimately reduced to wildcard arguments of class types. More precisely we restrict applications to wildcard arguments to reducible type constructors, where a type constructor is *reducible* if one of the following is true.

- The constructor is a reference to a class or trait
- The constructor is a type lambda of the form

  ```
  [X] -> B[... X ...]
  ```

  where `B` is a reference to a class or trait and `X` appears at most once, in argument position to `B`.
- The constructor is an alias of a reducible constructor.
- The constructor is an abstract type, and any bounds given for it are reducible constructors.

Example: Assume the declarations

```
type C[X] <: Iterable[X]
type M[X] = Map[X, X]
```

Then `C[_]` is legal because `C` is reducible but `M[_]` is illegal because `M` is irreducible.

The idea is that it is safe to beta-reduce an application of a type lambda of the form given above to a wildcard argument. For instance,

```
([X] -> C[X])[_]
```

is simply `C[_]`. On the other hand,

```
([X] -> M[X, X])[_]
```

is semantically *not* the same as `M[_, _]`. The former type implies a coupling between the two unknown type parameters which the latter type lacks.

The reducibility restriction does not seem to be very burdensome in practice. The dotty test suite, which includes Scala's standard collection library did not contain any occurrences of irreducible applications that would have to be rejected.

4.2 Implementation

The changes for supporting the direct representation are contained in pull request `#1343` of the `lampepfl/dotty` repository on GitHub. The base-line of that pull request is the refinement encoding presented in Section 6.

The changes can be summarized as follows.

- New syntax for type lambdas. The additional syntax is:

  ```
  Type  ::=
      HkTypeParamClause '->' Type
  HkTypeParamClause  ::=
      '[' HkTypeParam {',' HkTypeParam} ']'
  HkTypeParam   ::=
      {Annotation} ['+' | '-']
      (Id[HkTypeParamClause] | '_')
      TypeBounds
  ```

- Internal representations for type lambdas and higher-kinded applications as two new forms of types.
- A "smart" constructor for type application `C[T1, .., Tn]` that picks the member-based encoding if `C` is a class reference, beta-reduces if `C` is a type lambda, and returns a higher-kinded application otherwise.
- Extractors for type lambdas and type applications that work independently of the underlying representation.
- Implementation of subtyping and inference rules for type lambdas and applications as outlined above.

4.3 Comparison with scalac

The direct representation shares many characteristics with *scalac's* implementation of higher-kinded types, which was originally done by Adriaan Moors. In particular, the algorithms for subtyping and type inference are quite similar. But there are also differences to note.

In *scalac*, *all* forms of type applications are represented the same way. Type arguments are recorded as an additional list-valued field in a `TypeRef` node, which is one of the fundamental constructors with which the compiler represents types. Furthermore, *all* type definitions have a field that records the definition's type parameters. Type lambdas are not a primitive concept in *scalac*; the Scala community has instead settled on a rather elaborate encoding using structural types with type members and type projection, which bears some resemblance to the projection encoding in the next section.

By contrast, in *dotty* type parameters of classes are simply specially marked type members. For alias and abstract types, type parameters are expressed in the form of type lambdas. E.g., a source level definition like

```
type C[X] <: Iterable[X]
```

is represented in the equivalent form

```
type C <: [X] -> Iterable[X].
```

In summary, type parameters in *dotty* are a derived concept, not a fundamental one.

Type arguments are represented in *dotty* as refinements as long as the type constructor is a reference to a class or a trait. For type constructors that are abstract or alias types, there is a special type node called `HKApply` which has the type constructor and its arguments as fields.

4.4 Discussion

The direct encoding supports higher-kinded types in their full generality. Partial applications are supported through the introduction of type lambdas, which are notationally heavier than the solution of the simple encoding, but are much more legible than the workarounds using structural types and type projections in current Scala.

The direct representation is in a sense less elegant and economical than the simple encoding. It feels a bit awkward that type applications are encoded as refinements in the first-order case but remain as a primitive constructs in the higher-order case. On the other hand, this aligns well with the handling of wildcard arguments, which were the original motivation for encoding type applications as type member refinements. Wildcard arguments are expressible only if it can be guaranteed that they can be eventually reduced away. So in a sense, one of the main benefits of making the distinction between encoded and unencoded applications is that this obviates the need for existential types.

The conceptual and implementation cost of the direct representation suggests that it might be advantageous to study other encodings of higher-kinded types. Two such candidate encodings are presented in the next sections.

5. The Projection Encoding

The type projection approach was originally suggested by Adriaan Moors. It uses the following basic encodings.

- A type lambda `[X >: S <: U] -> T` is encoded as the refined type

  ```
  Lambda$I {
    type $hk0 >: S <: U
    type $Apply = [X := this.$hk0]T
  }
  ```

  This makes use of a family of synthetic base traits `Lambda$...`, one for each vector of variances of possible higher-kinded parameters. A suffix of `I` indicates a non-variant parameter, `P` (positive) a covariant parameter, and `N` (negative) a contravariant parameter. An n-ary base trait defines parameters `hk`i for $i = 1, ..., n$ with the given variances, as well as an abstract type member `$Apply`. For instance, the base trait

  ```
  trait Lambda$NP {
    type $hk0
    type $hk1
    type $Apply
  }
  ```

  is used for binary type lambdas where the first type parameter is contravariant and the second is covariant.

- An application of a non-variant higher kinded type `C` to an argument `T` is encoded as

  ```
  C { type $hk0 = T } # $Apply
  ```

  Covariant and contravariant type applications lead to refinements with upper and lower bounds instead.

- Beta reduction is supported by dereferencing the type projection. Indeed,

  ```
  ([X] -> T)[A]
  ```

  is encoded as

  ```
  Lambda$I {
    type $hk0
    type $Apply = [X := this.$hk0]T
  } {
    type $hk0 = A
  } # Apply
  ```

  which reduces to

  ```
  [this.$hk0 := A][X := this.$hk0]T
  ```

  which is equivalent to

  ```
  [X := A]T.
  ```

Ideally, an encoding of higher-kinded types into type members and refinements would be sufficiently expressive; an encoded term should be type-checkable in the base calculus without special provisions that account for the fact that types were originally higher-kinded. Unfortunately, there are a number of areas where higher-kinded types do shine through. To make, e.g. the standard Scala collections compile, all of the following tweaks are necessary:

1. `$Apply` refinements are covariant. If `T <: U` then

   ```
   S { type $Apply = T }
   <:  S { type $Apply = U }
   ```

   This subtyping relationship does not hold for ordinary type refinements. It would hold for upper bound refinements, of course. But we cannot model `$Apply` refine-

ments as upper bound refinements because that would lose beta reduction.

2. Encodings of type lambdas distribute over intersections and unions. For instance,

   ```
   Lambda$I { ... type $Apply = T } &
   Lambda$P { ... type $Apply = U }
   ```

   needs to be normalized to

   ```
   Lambda$I { ... type $Apply = T & U }
   ```

3. A subtype test of the encoded version of

   ```
   ([X1, ..., Xn] -> T) <: C
   ```

   where `C` is a class constructor is rewritten to:

   ```
   T <: C[X1, ..., Xn].
   ```

   Analogously, the subtype test of the encoded version of

   ```
   C <: ([X1, ..., Xn] -> T)
   ```

   is rewritten to

   ```
   C[X1, ..., Xn] <: T.
   ```

4. Inference of higher-kinded type parameters is handled using an algorithm analogous to the one described in Section 4.

One problematic aspect of the projection encoding is that generalized type projections have been shown to be unsound [7]. The known examples of unsoundness do not overlap with the images of the projection encoding, so it is conceivable that one could have a restricted form of type projection that permits the encoding of higher-kinded types and that is at the same time sound. But the rules for such a restricted form of type projection have not been worked out, and indeed the plan for future Scala is to allow only classes as prefixes of type projections. This can still model Java's inner classes (i.e. `C#I` is Scala's version of Java's inner class reference `C.I`), but it cannot model higher-kinded types which relies on the abstract type `$Apply` appearing in prefix positon of type projections.

5.1 Discussion

The projection encoding can model full higher-kinded types. It is based on two concepts already present in current Scala, type members and type projections. However, the latter concept is about to be phased out because it has been shown to be unsound.

The implementation overhead of the projection encoding was considerable and debugging was hard because encoded types can become quite large. But ultimately, it succeeded in representing all higher-kinded types in the standard library.

6. The Type Refinement Encoding

Whereas the type projection encoding makes use of an operator (type projection `#`) not covered in DOT but present in current Scala, the type refinement encoding uses general recursive types which are part of DOT but absent in Scala. The idea is as follows.

- A type lambda `[X >: S <: U] -> T` is encoded as the refined type

  ```
  [X := this.$hk0]T {
    type $hk0 >: S <: U
  }
  ```

- An application of a non-variant higher kinded type `C` to an argument `T` is encoded as

  ```
  C { type $hk0 = T }
  ```

  Covariant and contravariant type applications lead to refinements with upper and lower bounds instead.

- Beta reduction is a little bit problematic:

  ```
  ([X] -> T)[A]
  ```

  is encoded as

  ```
  [X := this.$hk0]T {
    type $hk0
  } {
    type $hk0 = A
  }
  ```

  This is equivalent to (i.e. both a subtype and a supertype of):

  ```
  [this.$hk0 := A]
     [X := this.$hk0]T {
        type $hk0 = A
     }
  ```

  which simplifies to

  ```
  [X := A]T { type $hk0 = A }
  ```

  The latter type is a subtype of the beta-reduced type `[X := A]T` but it is not a supertype, because of the occurrence of the additional refinement `{ type $hk0 = A }`. To make beta reduction work correctly, we have to add a "garbage collection" rule along the following lines.

$$\frac{T \text{ is a first-order type}}{T <: T\{\$hk_i = U\}}$$

To encode all higher-kinded types, the refinement encoding needs the full power of recursive types. For instance, the type

```
type Rep[T] = T
```

	Lines of code	Full hk types	Type lambdas	Full inference	Implementation effort
Simple	0	no	no	no	1 person-month
Projection [4]	719	yes	no	no	4 person-months
Refinement [8]	1216	yes	yes	no	1 person-month
Direct [6]	2000	yes	yes	yes	1 person-month

Table 1. Implementation characteristics

would lead to the encoding

```
type Rep = { z => z.$hk0; type $hk0 }
```

Here, the right hand side is a path-dependent recursive type, where "self" is represented by the variable `z`. Scala cannot currently express types like this, but DOT can. We have extended the *dotty* compiler to be able to cope with such general recursive types.

Like the projection encoding, the refinement encoding needed several tweaks in the compiler. The necessary changes are contained in pull request `#1282` of the `lampepfl/dotty` repository on GitHub.

The main changes necessary were, in addition to tweaks (2) - (4) of the type projection encoding:

1. Support for general recursive types, as outlined above.
2. Two "normalization functions" that essentially perform beta reduction. One of these (called `betaReduce`) was applied eagerly whenever a type application was formed; the other (`normalizeHkApply`) was applied every time the application was accessed.
3. A special case that disregards superfluous bindings of higher-kinded type parameters, as outlined in the garbage collection rule above.
4. A special case that disregards parameter bounds checking when comparing two encodings of type lambdas. The problem here is that naturally parameter bounds are contravariant whereas in the encoding they become member bounds, which are covariant. Disabling bounds checking for encodings of type lambdas thus avoids spurious type errors. Type soundness can still be guaranteed if one type-checks all type applications instead. In that case, type errors are simply reported reported later, on first-orer type formation. However, it turned out subsequently that checking all type applications, including in types inferred by the compiler is not very practical; so one might be better off enforcing the proper contravariant bounds relationship for type lambdas.

A recurring problem in the implementation of the refinement encoding was that circular types would arise during type simplification. An example of such a circular type is

```
C { type $hk0 = this.$hk0 }.
```

In theory, such circular types are harmless, but naive implementations of most type operations would send the compiler into an infinite loop. So cycles like these had to be detected and eliminated, which turned out to be difficult.

6.1 Discussion

The refinement encoding has the advantage that it is very closely integrated with DOT. It uses the full power of recursive types of DOT to model higher-kinded types. Unlike the projection encoding it does not need additional fundamental concepts like type projection whose status in future Scala is unclear. On the other hand, the abstraction presented by the refinement encoding is also leaky. Additional subtyping rules for garbage collection and type lambdas are needed, and the compiler needed a subtle combination of two type normalization rules. Also, one of these type normalization rules follows higher-kinded aliases when a type was applied, which leads to suboptimal type inference.

7. Comparison of Implementations

Table 1 gives some of the characteristics of the different implementations. The lines of code number gives approximate additional lines of code relative to the simple encoding. It includes whitespace, comments, and other documentation but excludes tests. The numbers are taken from the pull requests that implemented the proposals; Smaller changes to the different proposals that occurred after the initial pull requests are not taken into account.

The other columns in Table 1 indicate whether higher kinded types are supported in full generality, whether type lambdas are supported, and whether type inference is as complete as in current *scalac* with the inclusion of the fix to SI-2712.

The implementation of the simple encoding is smallest, but lacks all three of these properties. The refinement encoding is about 500 lines larger than the projection encoding, but includes syntactic support for type lambdas, and also includes several ameliorations in the handling of recursive types. The direct representation has the largest implementation footprint. On the other hand, it is the only one that supports type inference on a par with *scalac*.

The final column in Table 1 gives estimated implementation cost in (full-time) person months. These are rough estimates derived from personal recollections with the help of some GitHub archaeology. As all effort estimates, they have to be taken with a grain of salt. The projection encoding took

much longer than the others not just because of its implementation difficulties but also because it was our first attempt at implementing full higher-kinded types. Subsequent implementation schemes were implemented faster in part because of what was learned before.

8. Conclusion

The work on the four different implementations of higher-kinded types was done 2013-2016 in the context of the *dotty* compiler. Initially, *dotty* supported the simple encoding. Because of its lack of expressiveness this was discarded in favor of the projection encoding. Once it became clear that general type projection was unsound, we investigated the refinement encoding as an alternative to the projection encoding.

Neither encoding turned out to be fully satisfactory. Both were leaky in the sense that they demanded certain rules that applied specifically to constructs that resulted from encodings. Both also posed considerable difficulties for implementation and debugging. In retrospect, the biggest problem with the projection encoding was the size of the encoded types, which made diagnostics and debugging hard. The refinement encoding added somewhat less bulk, but suffered from the fact that cyclic bindings were often created inadvertently. Both encodings posed the problem that, being encodings, they were not reflected in static types. So the safety net of static typing was largely unavailable to the type checker itself.

In the end, *dotty* settled for a direct representation of higher-kinded types. This implementation was larger than the others, due to the fact that less typing infrastructure could be re-used. On the other hand, each of the higher-kinded constructs of type lambdas and type applications now was represented by its own static type, which was a big help in ensuring the correctness and completeness of the implementation.

In a sense, the direct representation gives an honest account of the additional implementation overhead caused by higher-kinded types. The overhead is non-negligible: about 2'000 lines compared to a total of about 28'000 lines which taken up by core data-structures and the type-checker.

In retrospect, we believe the simple encoding is an interesting alternative for a language that wants to provide most of the benefits of higher-kinded types at minimal cost to specification and implementation, provided one can arrive at a crisp definition of what is legal and what is not. But Scala is not that language, since it has a large installed code base that makes essential use of full higher-kinded types. The lesson learned from the work on the *dotty* compiler was that one is best off supporting full higher-kinded types directly. Encodings seem attractive at first for the code reuse they can provide, but in the end they cause more difficulties than they remove.

Acknowledgments

The *dotty* compiler has profited from the contributions of many people; Main contributors besides the authors include Felix Mulder, Ondřej Lhoták, Liu Fengyun, Vladimir Nikolayev, Samuel Grütter, Vera Salvis, Sébastien Doeraene, Jason Zaugg, and Nicolas Stucki. Adriaan Moors did the original implementation of higher-kinded types in *scalac* which informed our implementatio to no small degree. He as well as Rex Kerr, Daniel Spiewak, Sandro Stucki and Jason Zaugg provided important feedback on some of the implementations presented in this paper.

References

[1] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, pages 441–469, 2002. URL `http://link.springer.de/link/service/series/0558/bibs/2374/23740441.htm`.

[2] J. Mitchell and G. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.

[3] A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *Proc. OOPSLA*, pages 432–438, 2008.

[4] M. Odersky. Projection encoding of higher-kinded types, 2014. URL `https://github.com/lampepfl/dotty/pull/137`.

[5] M. Odersky. Compilers are databases. In *JVM Languages Summit*, 2015. URL `https://www.youtube.com/watch?v=WxyyJyB_Ssc`.

[6] M. Odersky. Direct representation of higher-kinded types, 2016. URL `https://github.com/lampepfl/dotty/pull/1343`.

[7] M. Odersky. Type projection is unsound, 2016. URL `https://github.com/lampepfl/dotty/issues/1050`.

[8] M. Odersky. Type refinement encoding of higher-kinded types, 2016. URL `https://github.com/lampepfl/dotty/pull/1282`.

[9] T. Rompf and N. Amin. Type Soundness for Dependent Object Types (DOT). *OOPSLA*, 2016. To appear.

[10] M. Sabin. SI-2712 add support for partial unification of type constructors, 2016. URL `https://github.com/scala/scala/pull/5102`.

[11] R. Tate, A. Leung, and S. Lerner. Taming wildcards in java's type system. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 614–627, 2011. .

[12] M. Torgersen, E. Ernst, C. P. Hansen, P. von der Ahé, G. Bracha, and N. M. Gafter. Adding wildcards to the java programming language. *Journal of Object Technology*, 3(11): 97–116, 2004. . URL `http://dx.doi.org/10.5381/jot.2004.3.11.a5`.