Polymorphic Function Types

A polymorphic function type is a function type which accepts type parameters. For example:

```
// A polymorphic method:
def foo[A](xs: List[A]): List[A] = xs.reverse

// A polymorphic function value:
val bar: [A] => List[A] => List[A]
// ^^^^^^^^^^^^^^^^^^^^^^^^^
// a polymorphic function type
       = [A] => (xs: List[A]) => foo[A](xs)
```

Scala already has *polymorphic methods*, i.e. methods which accepts type parameters. Method `foo` above is an example, accepting a type parameter `A`. So far, it was not possible to turn such methods into polymorphic function values like `bar` above, which can be passed as parameters to other functions, or returned as results.

In Scala 3 this is now possible. The type of the `bar` value above is

```
[A] => List[A] => List[A]
```

This type describes function values which take a type `A` as a parameter, then take a list of type `List[A]`, and return a list of the same type `List[A]`.

More details

Example Usage

Polymorphic function type are particularly useful when callers of a method are required to provide a function which has to be polymorphic, meaning that it should accept arbitrary types as part of its inputs.

For instance, consider the situation where we have a data type to represent the expressions of a simple language (consisting only of variables and function applications) in a strongly-typed way:

```
enum Expr[A]:
  case Var(name: String)
  case Apply[A, B](fun: Expr[B => A], arg: Expr[B]) extends Expr[A]
```

We would like to provide a way for users to map a function over all immediate subexpressions of a given `Expr`. This requires the given function to be polymorphic, since each subexpression may have a different type. Here is how to implement this using polymorphic function types:

```
def mapSubexpressions[A](e: Expr[A])(f: [B] => Expr[B] => Expr[B]): Expr[A] =
  e match
    case Apply(fun, arg) => Apply(f(fun), f(arg))
    case Var(n) => Var(n)
```

And here is how to use this function to *wrap* each subexpression in a given expression with a call to some `wrap` function, defined as a variable:

```
val e0 = Apply(Var("f"), Var("a"))
val e1 = mapSubexpressions(e0)(
  [B] => (se: Expr[B]) => Apply(Var[B => B]("wrap"), se))
println(e1) // Apply(Apply(Var(wrap),Var(f)),Apply(Var(wrap),Var(a)))
```

Relationship With Type Lambdas

Polymorphic function types are not to be confused with *type lambdas*. While the former describes the *type* of a polymorphic *value*, the latter is an actual function value *at the type level*.

A good way of understanding the difference is to notice that *type lambdas are applied in types, whereas polymorphic functions are applied in terms*: One would call the function `bar` above by passing it a type argument `bar[Int]` *within a method body*. On the other hand, given a type lambda such as `type F = [A] =>> List[A]`, one would call `F` *within a type expression*, as in `type Bar = F[Int]`.

‹ Depen... Enums ›

Contributors to this page

 pikinier20  BarkingBad  julienrf  odersky  michelou

 Master-Killer  b-studios  romanowski  LPTK