EPFL

# Type-Level Programming

Principles of Functional Programming

# Inductive Implicits (1)

An arbitrary number of implicit definitions can be combined until the search hits a “terminal” definition:

```
implicit def a: A = ...
implicit def aToB(implicit a: A): B = ...
implicit def bToC(implicit b: B): C = ...
implicit def cToD(implicit c: C): D = ...

implicitly[D]
```

# Inductive Implicits (2)

```
trait Nat
trait Z extends Nat
trait S[N <: Nat] extends Nat

implicit def zero: Z = null
implicit def one  (implicit zero: Z):       S[Z]       = null
implicit def two  (implicit one: S[Z]):     S[S[Z]]    = null
implicit def three(implicit two: S[S[Z]]): S[S[S[Z]]] = null

implicitly[S[S[S[Z]]]]
```

# Inductive Implicits (3)

```
trait Nat
trait Z extends Nat
trait S[N <: Nat] extends Nat

implicit def zero: Z = null
implicit def succ[N <: Nat](implicit n: N): S[N] = null

implicitly[S[S[S[Z]]]]
```

# Recursive Implicits

```
trait A
implicit def loop(implicit a: A): A = a

implicitly[A]
```

# Recursive Implicits

```
trait A
implicit def loop(implicit a: A): A = a

implicitly[A]

          ^
error: diverging implicit expansion for type A
starting with method loop
```

# Computing Types

```
trait Water
trait Ice

trait Melted[A, R]

object Melted {
  implicit def meltedIce: Melted[Ice, Water] = null
}

def meltedIce[R](implicit m: Melted[Ice, R]): R = ???

> def water = meltedIce
water: Water
```

# From Types to Values (1)

```
type ‘0‘ = Z
type ‘1‘ = S[‘0‘]
type ‘2‘ = S[‘1‘]
type ‘3‘ = S[‘2‘]
...

val x: ‘2‘ = ‘2‘
             ^
error: not found: value ‘2‘
```

# From Types to Values (2)

```
case class ValueOf[N <: Nat] private (get: Int)

object ValueOf {

  implicit def base: ValueOf[Z] = ValueOf(0)

  implicit def induc[A <: Nat](implicit h: ValueOf[A]): ValueOf[S[A]] =
    ValueOf(h.get + 1)

}
```

```
> implicitly[ValueOf[‘3‘]].get
res0: Int = 3
```

# Example: Sized Collections (1)

```
case class Sized[N <: Nat] private (elems: Seq[Int]) {

  def + (other: Sized[N]): Sized[N] =
    Sized(elems.zip(other.elems).map { case (x, y) => x + y })

}

def usage(xs: Sized[‘3‘], ys: Sized[‘3‘], zs: Sized[‘4‘]) = {
  xs + ys // OK
  xs + zs // Error: type mismatch;
          //         found   : Sized[‘4‘]
          //         required: Sized[‘3‘]
}
```

# Example: Sized Collections (2)

```
case class Sized[N <: Nat] private (elems: Seq[Int]) {

  def concat[M <: Nat](other: Sized[M]): Sized[???]

}
```

# Example: Sized Collections (3)

```
case class Sized[N <: Nat] private (elems: Seq[Int]) {

  def concat[M <: Nat, S <: Nat](
    other: Sized[M]
  )(implicit
    sum: Sum[N, M, S]
  ): Sized[S] =
    Sized(elems ++ other.elems)

}
```

# Example: Sized Collections (4)

```
trait Sum[A <: Nat, B <: Nat, R <: Nat]

object Sum {

  implicit def base[B <: Nat]: Sum[Z, B, B] = null

  implicit def induc[A <: Nat, B <: Nat, R <: Nat](implicit
    h: Sum[A, B, R]
  ): Sum[S[A], B, S[R]] = null

}
```

# Summary

In this video, we have seen:

- implicit definitions can be **inductive**
- types can be computed according to the result of the implicit search