



Conditional Given Definitions

Effective Programming in Scala

Comparing Sequences of Characters

Consider the ordering of String values:

► "abc" < "abd"?

a b c

a b d

Comparing Sequences of Characters

Consider the ordering of String values:

▶ "abc" < "abd"?

We compare the characters of each string, element-wise.

Comparing Sequences of Characters

Consider the ordering of String values:

▶ "abc" < "abd"?

We compare the characters of each string, element-wise.

Problem: Strings are sequences of characters, how to generalize the comparison process to sequences of any element type A?

Conditional Given Definitions (1)

This can be done only if the list elements have an ordering.

Given definitions can take type parameters and context parameters:

```
given orderingList[A](using ord: Ordering[A]): Ordering[List[A]] with  
  ...
```

orderingList is a *conditional* definition: an ordering for lists with elements of type A exists only if there is an ordering for A.

This sort of conditional behavior is best implemented with type classes. Subtyping and inheritance cannot express this: a class either inherits a trait or doesn't.

Conditional Given Definitions (2)

```
given orderingList[A](using ord: Ordering[A]): Ordering[List[A]] with
  def compare(xs0: List[A], ys0: List[A]) =
    (xs, ys) match
      case (Nil, Nil) => 0
      case (Nil, _)   => -1
      case (_, Nil)   => 1
      case (x :: xs1, y :: ys1) =>
        val c = ord.compare(x, y)
        if c != 0 then c else compare(xs1, ys1)

sort(List(List(1, 2, 3), List(1), List(1, 1, 3)))
// : List[List[Int]] = List(List(1), List(1, 1, 3), List(1, 2, 3))
```

Summoning Conditional Given Instances (1)

```
def sort[A](xs: List[A])(using ordering: Ordering[A]): List[A] = ...  
given orderingList[A](using ordering: Ordering[A]): Ordering[List[A]] with  
  ...  
  
val xss: List[List[Int]] = ...  
  
sort(xss)
```

A given instance for the outer type is constructed first and then its context parameters are filled in in turn.

Summoning Conditional Given Instances (2)

```
def sort[A](xs: List[A])(using ordering: Ordering[A]): List[A] = ...  
given orderingList[A](using ordering: Ordering[A]): Ordering[List[A]] with  
  ...  
  
val xss: List[List[Int]] = ...  
  
sort[List[Int]](xss)
```

A given instance for the outer type is constructed first and then its context parameters are filled in in turn.

Summoning Conditional Given Instances (3)

```
def sort[A](xs: List[A])(using ordering: Ordering[A]): List[A] = ...  
given orderingList[A](using ordering: Ordering[A]): Ordering[List[A]] with  
  ...
```

```
val xss: List[List[Int]] = ...
```

```
sort[List[Int]](xss)(using orderingList[Int])
```

A given instance for the outer type is constructed first and then its context parameters are filled in in turn.

Summoning Conditional Given Instances (4)

```
def sort[A](xs: List[A])(using ordering: Ordering[A]): List[A] = ...  
given orderingList[A](using ordering: Ordering[A]): Ordering[List[A]] with  
  ...
```

```
val xss: List[List[Int]] = ...
```

```
sort[List[Int]](xss)(using orderingList[Int](using Ordering.Int))
```

A given instance for the outer type is constructed first and then its context parameters are filled in in turn.

Deep Given Definitions

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

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

```
summon[D]
```

Deep Given Definitions (2)


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

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

```
summon[D](using cToD(using bToC(using aToB(using a))))
```

Recursive Given Definitions

```
trait A  
given loop(using a: A): A = a  
  
summon[A]
```



Recursive Given Definitions

```
trait A
given loop(using a: A): A = a
```

```
summon[A]
```

```
^^^^^^^^
```

```
error: no implicit argument of type A was found for parameter x of method summon.
      I found:
```

```
    loop(/* missing */summon[A])
```

But method loop produces a diverging implicit search when trying to match type A.

Exercise

Consider a program for managing an address book. We would like to sort the addresses by zip codes first and then by street name. Two addresses with different zip codes are ordered according to their zip code, otherwise (when the zip codes are the same) the addresses are ordered by street name.

```
type Address = (Int, String) // Zipcode, Street Name
val addresses: List[Address] = ...
sort(addresses)
```

- Implement an instance of Ordering for pairs of type (A, B), where both A and B have Ordering instances.

Exercise

- Implement an instance of Ordering for pairs of type (A, B), where both A and B have Ordering instances.

```
given orderingPair[A, B](  
  using ordA: Ordering[A], ordB: Ordering[B]): Ordering[(A, B)] with  
  def compare(x: (A, B), y: (A, B)) =  
    val c = ordA.compare(x(0), y(0))  
    if c != 0 then c else ordB.compare(x(1), y(1))
```


Summary

- ▶ given definitions can also take type parameters and context parameters,
- ▶ an arbitrary number of given definitions can be chained until a terminal definition is reached,
- ▶ this allows the compiler to summon complex pieces of logic based on type information, e.g.:

```
summon[Ordering[(Int, List[String])]]
```