Using Clauses

Effective Programming in Scala

Reminder: General `sort` Operation

```
def sort[A](xs: List[A])(ord: Ordering[A]): List[A] = ...
```

Problem: passing around `Ordering` arguments is cumbersome.

```
sort(xs)(Ordering.Int)
sort(ys)(Ordering.Int)
sort(strings)(Ordering.String)
```

Sorting a `List[Int]` always uses the same `Ordering.Int` argument, sorting a `List[String]` always uses the same `Ordering.String` argument, and so on...

Let the Compiler Write Our Program

So, we would like to just write:

```
sort(xs)
```

And let the compiler pass the ordering argument for us.

To achieve this we need to do two things:

1. let the compiler know that we expect it to pass the argument,
2. provide candidate values for such arguments.

This lecture shows how to address the first point.

Using Clauses

We let the compiler know that we expect it to pass the `ord` argument by making it a **context parameter**.

```
def sort[A](xs: List[A])(using ord: Ordering[A]): List[A] = ...
```

Then calls to `sort` can omit the `ord` argument:

```
sort(xs)
sort(ys)
sort(strings)
```

The compiler infers the argument value based on its expected type.

Type and Term Inference

```
def sort[A](xs: List[A])(using ord: Ordering[A]): List[A] = ...

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

Type and Term Inference

```
def sort[A](xs: List[A])(using ord: Ordering[A]): List[A] = ...

val xs: List[Int] = ...

sort(xs)
```

Type and Term Inference

```scala
def sort[A](xs: List[A])(using ord: Ordering[A]): List[A] = ...

val xs: List[Int] = ...

sort(xs)

sort[Int](xs)
```

Type and Term Inference

```
def sort[A](xs: List[A])(using ord: Ordering[A]): List[A] = ...

val xs: List[Int] = ...

sort(xs)

sort[Int](xs)
```

In this case, the type of `ord` is `Ordering[Int]`.

Type and Term Inference

```
def sort[A](xs: List[A])(using ord: Ordering[A]): List[A] = ...

val xs: List[Int] = ...

sort(xs)

sort[Int](xs)
```

In this case, the type of `ord` is `Ordering[Int]`.

```
sort[Int](xs)(using Ordering.Int)
```

(We will explain later the mechanism used by the compiler to find the value `Ordering.Int`)

Explicit Argument

Note that it is still possible to explicitly pass an argument by using a `using` argument clause, which can be useful to provide a non-default value:

```
sort(xs)(using Ordering.Int.reverse)
```

But the argument is usually left out.

Using Clauses Syntax Reference

Multiple context parameters can be in a `using` clause:

```
def f(x: Int)(using a: A, b: B) = ...
f(x)(using a, b)
```

Or, there can be several `using` clauses in a row:

```
def f(x: Int)(using a: A)(using b: B) = ...
```

`using` clauses can also be freely mixed with regular parameters:

```
def f(x: Int)(using a: A)(y: Boolean)(using b: B) = ...
```

Anonymous Using Clauses

Parameters of a `using` clause can be anonymous:

```
def sort[A](xs: List[A])(using Ordering[A]): List[A] =
  ...
  ... merge(sort(fst), sort(snd))
  ...

def merge[A](xs: List[A], ys: List[A])(using Ordering[A]): List[A] = ...
```

This is useful if the body of `sort` does not mention `ord` at all, but simply passes it on as a context argument to further methods.

Context Bounds

There is a shorter, alternate syntax for context parameters:

```scala
def sort[A: Ordering](xs: List[A]): List[A] = ...
```

We say that the type parameter `A` has one **context bound**: `Ordering`.

This is equivalent to the following signature:

```scala
def sort[A](xs: List[A])(using Ordering[A]): List[A]
```

Summary

Parameters marked by a `using` clause can be left out at the call site.

The compiler infers their value based on their expected type.