Using Clauses and Given Instances

Principles of Functional Programming

Martin Odersky and Julien Richard-Foy

Using Clauses

An implicit parameter is introduced by a `using` parameter clause:

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

A matching explicit argument can be passed in a `using` argument clause:

```
sort(strings)(using Ordering.String)
```

But the argument can also be left out (and it usually is).

If the argument is missing, the compiler will infer one from the parameter type.

```
sort(strings)
```

Using Clauses Syntax Reference

Multiple 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) = ...
f(x)(using a)(y)(using b)
```

Anonymous Using Clauses

Parameters of a `using` clause can be anonymous:

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

def merge[T](xs: List[T])(using Ordering[T]): List[T] = ...
```

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

Anonymous Using Clauses

Parameters of a `using` clause can be anonymous:

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

def merge[T](xs: List[T])(using ord: Ordering[T]): List[T] = ...
```

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

Context Bounds

Sometimes one can go further and replace the using clause with a context bound for a type parameter.

Instead of:

```
def printSorted[T](as: List[T])(using Ordering[T]) =
  println(sort(as))
```

Context Bounds

Sometimes one can go further and replace the using clause with a context bound for a type parameter.

With a context bound:

```
def printSorted[T: Ordering](as: List[T]) =
  println(sort(as))
```

Context Bounds

Sometimes one can go further and replace the using clause with a context bound for a type parameter.

With a context bound:

```
def printSorted[T: Ordering](as: List[T]) =
  println(sort(as))
```

More generally, a method definition such as:

$$\mathbf{def}\ f[T : U_1\ \ldots\ : U_n](ps) : R = \ldots$$

is expanded to:

$$\mathbf{def}\ f[T](ps)(\mathbf{using}\ U_1[T], \ldots, U_n[T]) : R = \ldots$$

Given Instances

For the previous example to work, the `Ordering.Int` definition must be a `given` instance:

```
object Ordering:

  given Int: Ordering[Int] with
    def compare(x: Int, y: Int): Int =
      if x < y then -1 else if x > y then 1 else 0
```

This code defines a given instance of type `Ordering[Int]`, named `Int`.

Anonymous Given Instances

Given instances can be anonymous. Just omit the instance name:

```
given Ordering[Double] with
  def compare(x: Int, y: Int): Int = ...
```

The compiler will synthesize a name for an anonymous instance:

```
given given_Ordering_Double: Ordering[Double] with
  def compare(x: Int, y: Int): Int = ...
```

Summoning an Instance

One can refer to a (named or anonymous) instance by its type:

```
summon[Ordering[Int]]
summon[Ordering[Double]]
```

These expand to:

```
Ordering.Int
Ordering.given_Ordering_Double
```

`summon` is a predefined method. It can be defined like this:

```
def summon[T](using x: T) = x
```

Implicit Parameter Resolution

Say, a function takes an implicit parameter of type `T`.

The compiler will search a *given instance* that:

- has a type compatible with `T`,
- is visible at the point of the function call, or is defined in a companion object *associated* with `T`.

If there is a single (most specific) instance, it will be taken as actual arguments for the inferred parameter.

Otherwise it's an error.

Given Instances Search Scope

The search for a given instance of type `T` includes:

- all the given instances that are visible (inherited, imported, or defined in an enclosing scope),
- the given instances found in a companion object *associated* with `T`.

The definition of *associated* is quite general. Besides the companion object of a class itself, the compiler will also consider

- companion objects associated with any of `T`'s inherited types
- companion objects associated with any type argument in `T`
- if `T` is an inner class, the outer objects in which it is embedded.

Companion Objects Associated With a Queried Type

If the compiler does not find a given instance matching the queried type `T` in the lexical scope, it continues searching in the companion objects associated with `T`.

Consider the following hierarchy:

```
trait Foo[T]
trait Bar[T] extends Foo[T]
trait Baz[T] extends Bar[T]
trait X
trait Y extends X
```

If a given instance of type `Bar[Y]` is required, the compiler will look into the companion objects `Bar`, `Y`, `Foo`, and `X` (but not `Baz`).

Importing Given Instances

Since given instances can be anonymous, how can they be imported?

In fact, there are three ways to import a given instance.

1. By-name:

```scala
import scala.math.Ordering.Int
```

2. By-type:

```scala
import scala.math.Ordering.{given Ordering[Int]}
import scala.math.Ordering.{given Ordering[?]}
```

3. With a wildcard:

```scala
import scala.math.given
```

Since the names of givens don't really matter, the second form of import is preferred since it is most informative.

Exercise

```
val xs = List(3, 1, 2)
sort(xs)
```

In the above example of the `sort` method call, where does the compiler find the given instance of type `Ordering[Int]`?

o In the enclosing scope

o Via a given import

o In a companion object associated with the type `Ordering[Int]`

Exercise

```
val xs = List(3, 1, 2)
sort(xs)
```

In the above example of the `sort` method call, where does the compiler find the given instance of type `Ordering[Int]`?

o In the enclosing scope

o Via a given import

x In a companion object associated with the type `Ordering[Int]`

- The given instance is found in the `Ordering` companion object

No Given Instance Found

If there is no available given instance matching the queried type, an error is reported:

```
scala> def f(using n: Int) = ()
scala> f
       ^
error: no implicit argument of type Int was found for parameter n of method f
```

Ambiguous Given Instances

If more than one given instance is eligible, an *ambiguity* is reported:

```
trait C:
  val x: Int
given c1: C with
  val x = 1
given c2: C with
  val x = 2

f(using c: C) = ()
f
^
error: ambiguous implicit arguments:
both value c1 and value c2
match type C of parameter c of method f
```

Priorities

Actually, several given instances matching the same type don't generate an ambiguity if one is *more specific* than the other.

In essence, a definition

```
given a: A
```

is more specific than a definition

```
given b: B
```

if:

- `a` is in a closer lexical scope than `b`, or
- `a` is defined in a class or object which is a subclass of the class defining `b`, or
- type `A` is a generic instance of type `B`, or
- type `A` is a subtype of type `B`.

Priorities: Example (1)

Which given instance is summoned here?

```scala
class A[T](x: T)
given universal[T](using x: T): A[T](x)
given specific: A[Int](2)

summon[A[Int]]
```

Priorities: Example (2)

Which given instance is summoned here?

```
trait A:
  given ac: C
trait B extends A:
  given bc: C
object O extends B:
  val x = summon[C]
```

Priorities: Example (3)

Which given instance is summoned here?

```scala
given ac: C
def f() =
  given b: C
  def g(using c: C) = ()

  g
```

Summary

In this lecture we have introduced a way to do *type-directed programming*, with the help of a language mechanism that infers *values* from *types*.

There has to be a *unique* (most specific) given instance matching the queried type for it to be used by the compiler.

Given instances are searched in the enclosing *lexical scope* (imports, parameters, inherited members) as well as in the companion objects associated with the queried type.