Type Classes

CS-214 Software Construction

Type Classes

In the previous lectures we have seen a particular pattern of code:

```
trait Ordering[A]:
  def compare(x: A, y: A): Int

object Ordering:
  given Int: Ordering[Int] with
    def compare(x: Int, y: Int) =
      if x < y then -1 else if x > y then 1 else 0
  given String: Ordering[String] with
    def compare(x: String, y: String) = s.compareTo(t)
  given List: [T](using Ordering[T]): Ordering[List[T]] with
    def compare(x: List[T], y: List[T]) = ...
```

Type Classes

We say that `Ordering` is a *type class*.

In Scala, a type class is a generic trait that comes with given instances for type instances of that trait.

E.g., in the `Ordering` example, we have given instances for

- `Ordering[Int]`
- `Ordering[String]`
- `Ordering[List[T]]` for all `T`'s that have `Ordering`s.

Other instances can be defined, either in the `Ordering` object or elsewhere.

Ad-Hoc Polymorphism

Type classes provide yet another form of *polymorphism*:

The `sort` method can be called with lists containing elements of any type `A` for which there is a given instance of type `Ordering[A]`.

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

At compilation-time, the compiler resolves the specific `Ordering` implementation that matches the type of the list elements.

This means that the a type `Ordering[A]` has different implementations for different types `A`.

This is sometimes called *ad-hoc polymorphism*

Context Bounds

The `(using TypeClass[T])` syntax is so common that Scala has an abbreviation for it:

Instead of:

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

we can write

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

In words: `sort` works for all types `T` that have an `Ordering`.

Context Bounds

The `(using TypeClass[T])` syntax is so common that Scala has an abbreviation for it:

Instead of:

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

we can write

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

In words: `sort` works for all types `T` that have an `Ordering`.

Translation of Context Bounds

More generally, a method definition such as:

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

is expanded to:

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

Context Bounds for Conditional Instances

Context bounds can also be used in conditional given instances:

```
given listOrdering[T: Ordering]: Ordering[List[T]] with ...

given pairOrdering[A: Ordering, B: Ordering]: Ordering[(A, B)] with ...

given [A: Pickler, B: Pickler]: Pickler[(A, B)] with ...
```

Exercise

Implement an instance of the `Ordering` typeclass for the `Rational` type.

```
case class Rational(numer: Int, denom: Int)
```

Reminder:

let $q = \frac{num_q}{denom_q}$, $r = \frac{num_r}{denom_r}$,

$q < r \Leftrightarrow \frac{num_q}{denom_q} < \frac{num_r}{denom_r} \Leftrightarrow num_q \times denom_r < num_r \times denom_q$

Exercise

Implement an instance of the `Ordering` typeclass for the `Rational` type.

```
case class Rational(numer: Int, denom: Int)
```

Reminder:

let $q = \frac{num_q}{denom_q}$, $r = \frac{num_r}{denom_r}$,

$q < r \Leftrightarrow \frac{num_q}{denom_q} < \frac{num_r}{denom_r} \Leftrightarrow num_q \times denom_r < num_r \times denom_q$

```
given Ordering[Rational] with
  def compare(x: Rational, y: Rational) =
    Ordering.Int.compare(x.numer * y.denom, y.numer * x.denom)
```

Digression: Retroactive Extension

It is worth noting that we were able to implement the `Ordering[Rational]` instance without changing the `Rational` class definition.

Type classes support *retroactive* extension: the ability to extend a data type with new operations without changing the original definition of the data type.

In this example, we have added the capability of comparing `Rational` numbers.

Caveat: Such retroactive extensions need to be defined or imported explicitly at the points where they are used, since we can't put them in companion objects.

Typeclasses in Other Languages

Languages besides Scala that support type classes

- Haskell (that's where the name comes from)
- Rust, under the name *traits* (Scala traits are impl traits or dyn traits in Rust)
- Swift, under the name *protocols*
- Lean 4

Type Classes and Extension Methods

Like any trait, a type class trait may define extension methods.

For, instance, the `Ordering` trait would usually contain comparison methods like this:

```
trait Ordering[A]:

  def compare(x: A, y: A): Int

  extension (x: A)
    def <  (y: A): Boolean = compare(x, y) < 0
    def <= (y: A): Boolean = compare(x, y) <= 0
    def >  (y: A): Boolean = compare(x, y) > 0
    def >= (y: A): Boolean = compare(x, y) >= 0
```

Visibility of Extension Methods

Extension methods on a type class trait are visible whenever a given instance for the trait is available.

For instance one can write:

```
def merge[T: Ordering](xs: List[T], ys: List[T]): Boolean = (xs, ys) match
  case (Nil, _) => ys
  case (_, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y then x :: merge(xs1, ys)
    else y :: merge(xs, ys1)
```

- There's no need to name and import the `Ordering` instance to get access to the extension method < on operands of type `T`.
- We have an `Ordering[T]` instance in scope, that's where the extension method comes from.

Summary

Type classes provide a way to turn types into values.

Unlike class extension, type classes

- can be defined at any time without changing existing code,
- can be conditional.

In Scala, type classes are constructed from parameterized traits and given instances.

Type classes give rise to a new kind of polymorphism, which is sometimes called *ad-hoc* polymorphism.

This means that the a type `TC[A]` has different implementations for different types `A`.

Abstract Algebra and Type Classes

CS-214 Software Construction

Doing Abstract Algebra with Type Classes

Type classes let one define concepts that are quite abstract, and that can be instantiated with many types. For instance:

```
trait SemiGroup[T]:
  extension (x: T) def combine (y: T): T
```

This models the algebraic concept of a semigroup with an associative operator `combine`.

Doing Abstract Algebra with Type Classes

Type classes let one define concepts that are quite abstract, and that can be instantiated with many types. For instance:

```
trait SemiGroup[T]:
  extension (x: T) def combine (y: T): T
```

This models the algebraic concept of a semigroup with an associative operator `combine`.

We can then define methods that work for all semigroups. For instance:

```
def reduce[T: SemiGroup](xs: List[T]): T =
  xs.reduceLeft(_.combine(_))
```

Type Class Hierarchies

Algebraic type classes often form natural hierarchies. For instance, a *monoid* is defined as a semigroup with a left and right unit element.

Here's its natural definition:

```
trait Monoid[T] extends SemiGroup[T]:
  def unit: T
```

Exercise

Generalize `reduce` to work on lists of `T` where `T` has a `Monoid` instance such that it also works for empty lists.

Exercise

Generalize `reduce` to work on lists of `T` where `T` has a `Monoid` instance such that it also works for empty lists.

```
def reduce[T](xs: List[T])(using m: Monoid[T]): T =
  xs.foldLeft(m.unit)(_.combine(_))
```

Using Context Bounds

In the previous example we had to pass an explicitly named type class instance `m: Monoid[T]` to `reduce`, so that we could refer to `m.unit`.

One could alternatively use a context bound and a `summon`.

```
def reduce[T: Monoid](xs: List[T]): T =
  xs.foldLeft(summon[Monoid[T]].unit)(_.combine(_))
```

Using Context Bounds

In the previous example we had to pass an explicitly named type class instance `m: Monoid[T]` to `reduce`, so that we could refer to `m.unit`.

One could alternatively use a context bound and a `summon`.

```
def reduce[T: Monoid](xs: List[T]): T =
  xs.foldLeft(summon[Monoid[T]].unit)(_.combine(_))
```

Aside: Scala 3.6 allows to name context bounds, so that one does not need the `summon`:

```
def reduce[T: Monoid as m](xs: List[T]): T =
  xs.foldLeft(m.unit)(_.combine(_))
```

Multiple Typeclass Instances

It's possible to have several given instances for a typeclass/type pair. For instance, `Int` could be a `Monoid` in (at least) two ways:

- with + as `combine` and `0` as `unit`, or
- with $*$ as `combine` and `1` as `unit`.

```
given sumMonoid: Monoid[Int] with
  extension (x: Int) def combine(y: Int) : Int = x + y
  def unit: Int = 0

given prodMonoid: Monoid[Int] with
  extension (x: Int) def combine(y: Int) : Int = x * y
  def unit: Int = 1
```

One then should explicitly import one or the other.

Typeclass Laws

Algebraic type classes are not just defined by their type signatures but also by the laws that hold for them.

For example, any given instance of `Monoid[T]` should satisfy the laws:

```
x.combine(y).combine(z)  ==  x.combine(y.combine(z))
        unit.combine(x)  ==  x
        x.combine(unit)  ==  x
```

where `x`, `y`, `z` are arbitrary values of type `T` and `unit = Monoid.unit[T]`.

The laws can be verified either by a formal or informal proof, or by testing them.

A good way to test that an instance is *lawful* is using randomized testing with a tool like ScalaCheck.

Context Passing

CS-214 Software Construction

Two Uses of Givens

We have seen two broad classes of how givens are used:

- as type classes, or
- for context passing.

Type classes are about *type instances of generic traits*. E.g.:

- What is the definition of `TC[A]` for the type class trait `TC` and the type argument `A`? If we want to make `A` a type parameter, we need an implicit parameter to go with it.

Passing context is often about implicitly parameterizing over values that can have a simple type, to answer the question

- What is the currently valid definition of type `T`?

Use Cases of Context Passing

Passing a piece of the context as an implicit parameter of a certain type is quite common.

For instance, we might want to propagate implicitly

- the current configuration,
- the available set of capabilities,
- the security level in effect,
- the layout scheme to render some data,
- The persons that have access to some data.

Case Study: A Conference Management System

Let's say we design a system to discuss papers submitted to a conference.

- The papers have already been given a score by the reviewers.
- To discuss, reviewers need to see various pieces of information about the papers.
- Some reviewers are also authors of papers.
- An author of a paper should never see at this phase the score the paper received from the other reviewers.

Consequence: Every query of the conference needs to know who is seeing the results of the operation and this needs to be propagated.

For a given toplevel query the set of persons seeing its results will largely stay the same.

But it can change, for instance when a reviewer *delegates* part of the task to another person.

Outline

```scala
case class Person(name: String)
case class Paper(title: String, authors: List[Person], body: String)

object ConfManagement:
  type Viewers = Set[Person]

  class Conference(ratings: (Paper, Int)*):
    private val realScore = ratings.toMap

    def papers: List[Paper] = ratings.map(_._1).toList

    def score(paper: Paper, viewers: Viewers): Int =
      if paper.authors.exists(viewers.contains) then -100
      else realScore(paper)
```

Outline ctd

```
    def rankings(viewers: Viewers): List[Paper] =
      papers.sortBy(score(_, viewers)).reverse

    def ask[T](p: Person, query: Viewers => T) =
      query(Set(p))

    def delegateTo[T](p: Person, query: Viewers => T)(viewers: Viewers): T =
      query(viewers + p)
  end Conference
end ConfManagement
```

- If one of the viewers is also an author if the paper, the score is *masked*, returning -100 instead of the real score.
- The same masking also has to be done in derived operations, such as `rankings`.

Example Dataset

```
val Smith  = Person("Smith")
val Peters = Person("Peters")
val Abel   = Person("Abel")
val Black  = Person("Black")
val Ed     = Person("Ed")

val conf = Conference(
  Paper("How to grow beans", List(Smith, Peters), "...") -> 92,
  Paper("Organic gardening", List(Abel, Peters), "...") -> 83,
  Paper("Composting done right", List(Black, Smith), "...") -> 99,
  Paper("The secret life of snails", authors = List(Ed), "...") -> 77
)
```

Example Query

Which authors have at least two papers with a score over 80?

```
def highlyRankedProlificAuthors(asking: Person): Set[Person] =
  def query(viewers: Viewers): Set[Person] =
    val highlyRanked =
      conf.rankings(viewers).takeWhile(conf.score(_, viewers) > 80).toSet
    for
      p1 <- highlyRanked
      p2 <- highlyRanked
      author <- p1.authors
      if p1 != p2 && p2.authors.contains(author)
    yield author
  conf.ask(asking, query)
```

The answer depends on who is asking!

Tamper-Proofing

Problem: So far passing viewers is a *convention*.

Nothing prevents just passing the empty set of viewers to a query.

```
conf.rankings(Set()).takeWhile(conf.score(_, Set()) > 80)
```

Fix: Make the `Viewers` type alias *opaque*:

```
opaque type Viewers = Set[Person]
```

Opaque Type Aliases

Given an opaque type alias such as

```
object ConfManagement:
  opaque type Viewers = Set[Person]
```

the equality `Viewers = Set[Person]` is known only within the scope where the alias is defined. (in this case, within the `ConfManagement` object)

Everywhere else `Viewers` is treated as a separate, abstract type.

Why Does This Help Against Tampering?

When asking a query, we have to pass a `Viewers` set to the conference management methods.

But `Viewers` is an unknown abstract type; hence there is no way to create a `Viewers` instance outside the `ConfManagement` object.

So the only way to get a viewers value is in the parameter of a query, where the conference management system provides the actual value.

Therefore, in

```
conf.rankings(viewers).takeWhile(conf.score(_, viewers) > 80).toSet
```

we are *forced* to pass `viewers` on to `rankings` and `score` since that's the only `Viewers` value we have access to.

Caveat: This assumes that queries are not nested, since otherwise an inner query could access the `viewers` parameter of an outer one)

Discussion

Back to the conference management code:

- One downside is that we have to pass `viewers` arguments along everywhere they are needed.
- This seems pointless, since *by design* there is only a single value we could pass!
- It also quickly gets tedious as the codebase grows.
- Can't this be automated?

Discussion

Back to the conference management code:

- One downside is that we have to pass `viewers` arguments along everywhere they are needed.
- This seems pointless, since *by design* there is only a single value we could pass!
- It also quickly gets tedious as the codebase grows.
- Can't this be automated?

Of course: Just use implicit parameters.

Using using Clauses

```
class Conference(ratings: (Paper, Int)*):
  ...
  def score(paper: Paper)(viewers: Viewers): Int =
    if paper.authors.exists(viewers.contains) then -100
    else realScore(paper)
  def rankings(viewers: Viewers): List[Paper] =
    papers.sortBy(score(_)(viewers)).reverse
  def delegateTo[T](p: Person, query: Viewers => T)(viewers: Viewers): T =
    query(viewers + p)
  ...
```

```
conf.rankings(viewers).takeWhile(conf.score(_, viewers) > 80).toSet
```

Using using Clauses

```
class Conference(ratings: (Paper, Int)*):
  ...
  def score(paper: Paper)(using viewers: Viewers): Int =
    if paper.authors.exists(viewers.contains) then -100
    else realScore(paper)
  def rankings(using viewers: Viewers): List[Paper] =
    papers.sortBy(score(_)).reverse
  def delegateTo[T](p: Person, query: Viewers => T)(using viewers: Viewers): T =
    query(viewers + p)
  ...
```

```
conf.rankings.takeWhile(conf.score(_) > 80).toSet
```

Another Benefit of Opacity

The implicit parameters are of type `Viewers`, which is an opaque type alias.

This has another benefit: Since outside `ConfManagement`, `Viewers` is a type different from all others , there's no chance to connect `Viewers` implicit parameters with given instances for other types.

On the other hand, if `Viewers` was a regular type alias of `Set[Person]` we might accidentally have given instances for other sets of persons in scope, which would then be eligible candidates for `Viewers` parameters.

Be Specific

Morale: Given instances should have specific types and/or be local in scope.

For example, this is a terrible idea:

```
given Int = 1
def f(x: Int)(using delta: Int) = x + delta
```

Never use a common type such as `Int` or `String` as the type of a globally visible given instance!

Context Function Types

CS-214 Software Construction

(Additional material, not required for the course)

Repetitive Using Clauses

In last version of the conference management system of the last session we got rid of explicit `Viewers` arguments.

But we still need explicit `using` parameter clauses.

```
def score(paper: Paper)(using Viewers): Int = ...
def rankings(using Viewers): List[Paper] = ...
def delegateTo(p: Person, query: Viewers => T)(using Viewers): T = ...
```

Can we get rid of these as well?

Lambdas With Using Clauses

Let's massage the definition of `rankings` a bit:

```
def rankings = (viewers: Viewers) =>
  papers.sortBy(score(_, viewers)).reverse
```

Lambdas With Using Clauses

Let's massage the definition of `rankings` a bit:

```
def rankings = (viewers: Viewers) ?=>
  papers.sortBy(score(_, viewers)).reverse
```

The `?` signifies that we want the parameter `viewers` be implicit so that its arguments can be inferred.

What is its type?

Lambdas With Using Clauses

Let's massage the definition of `rankings` a bit:

```
def rankings = (viewers: Viewers) ?=>
  papers.sortBy(score(_, viewers)).reverse
```

The `?` signifies that we want the parameter `viewers` be implicit so that its arguments can be inferred.

What is its type?

- For a normal anonymous function it would be:
  `Viewers => List[Paper]`
- For an anonymous functions with a `using` clause it is:
  `Viewers ?=> List[Paper]`

Context Function Types

`Viewers ?=> List[Paper]` is called a *context function type*.

There are two typing rules involving such types.

1. Context functions get their arguments inferred just like methods with using clauses. In

   ```
   val f: A ?=> B
   given a: A
   f
   ```

   the expression `f` expands to `f(using a)`.

Context Function Types

`Viewers ?=> List[Paper]` is called a *context function type*.

There are two typing rules involving such types.

1. Context functions get their arguments inferred just like methods with using clauses. In

   ```
   val f: A ?=> B
   given a: A
   f
   ```

   the expression `f` expands to `f(using a)`.
2. Context functions get created on demand.
   If the expected type of an expression `b` is `A ?=> B`, then `b` expands to the anonymous function `(_: A) ?=> b`.

Example Application

Let's use context function types in our conference management system.

First, introduce a type alias

```
type Viewed[T] = Viewers ?=> T
```

This is just for conciseness; `Viewed[T]` is easier to read than `Viewers ?=> T` and it expresses the point we want to make.

Example Application (2)

Now, perform the apply two changes:

1. Replace every method signature ending in

   ```
   (using Viewers): SomeType
   ```

   with

   ```
   : Viewed[SomeType]
   ```

Example Application (2)

Now, perform the apply two changes:

1. Replace every method signature ending in

   ```
   (using Viewers): SomeType
   ```

   with

   ```
   : Viewed[SomeType]
   ```

2. Replace function type parameter

   ```
   query: Viewers => SomeType
   ```

   with

   ```
   query: Viewed[SomeType]
   ```

Trade Types for Type Parameters

Implicit Parameters in using clauses trade *types* for *terms*:

- The developer writes down the required type of the parameter.
  The compiler infers an expression (i.e. a term) for it.

Trade Types for Type Parameters

Implicit Parameters in using clauses trade *types* for *terms*:

- The developer writes down the required type of the parameter.
  The compiler infers an expression (i.e. a term) for it.

Context Function Types go one step further. They trade types for parameters.

- The developer writes down the return type of the method.
  The compiler infers one or more method parameters that match the type.