# Implicit Function Types

Principles of Functional Programming

# 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]`

# Implicit Function Types

`Viewers ?=> List[Paper]` is called an *implicit function type*.

There are two typing rules involving such types.

1. Implicit 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)`.

# Implicit Function Types

`Viewers ?=> List[Paper]` is called an *implicit function type*.

There are two typing rules involving such types.

1. Implicit 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. Implicit 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 implicit 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.

Implicit 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.

# Abstracting over Context Abstractions

Another way to look at it is to see implicit function types, as *second degree context abstractions*.

- Implicit parameters in `using` clauses abstract over the context at the call site.
  They are first-degree context abstractions.
- Implicit function types allow to abstract over using clauses (in the original sense: they allow to introduce a name such as `Viewed` that can be used instead of writing explicit parameter clauses).
- So, together with type aliases, they enable abstractions of context abstractions.