Context Passing

Principles of Functional Programming

Context Passing vs Type Classes

Type classes are about *combinations of types*. 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.

On the other hand, there are also uses for abstracting over values of a simple type, asking

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

Example: Execution contexts

To do computations in parallel, runtimes need *thread schedulers*.

There's usually a default scheduler, but it should be possible to override that choice in parts of the code.

How are schedulers propagated?

In Scala, they are embedded in values of types `ExecutionContext`. The default is:

```
given global as ExecutionContext = ForkJoinContext()
```

This defines the execution context `global` as an alias of an existing value (i.e. a freshly created `ForkJoinContext`)

The evaluation of `ForkJoinContext` is done lazily: the `ForkJoinContext` is created the first time `global` is used.

Propagating Execution Contexts

Execution contexts rarely change, but they should be changeable everywhere.

This is a poster-child for implicit parameters.

```
def processItems(...)(using ExecutionContext) = ...
```

Other Use Cases

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.

Example: 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

```
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.

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):
    query(viewers + p)
  ...


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

Be Specific!

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.

Morale: Always make sure that the types of implicit parameters are very specific.

For example, this is a terrible idea:

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

Never use a common type such as `Int` or `String` as the type of an implicit parameter.