Implicit Function Types

Principles of Functional Programming

Martin Odersky

Con-text

"what comes with the text, but is not in the text"

Context is all around us

- the current configuration
- the current scope
- the meaning of “$<$” on this type
- the user on behalf of which the operation is performed
- the security level in effect

Traditional ways to express context

Globals

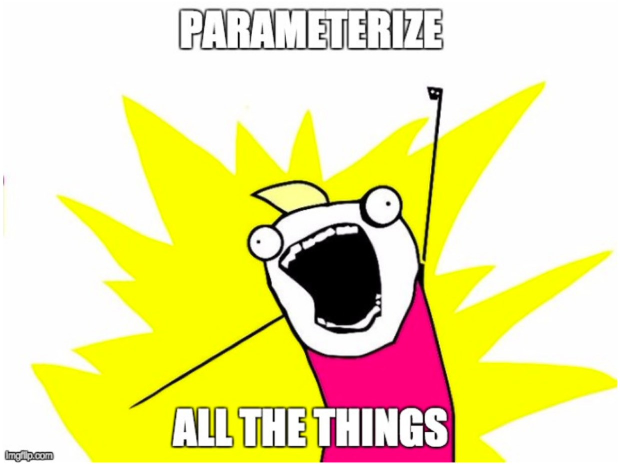
- rigid if immutable,
- unsafe if mutable

Monkey patching

Dependency injection

- (e.g. Spring, Guice)
- most commonly: inject constructor parameters

The Functional Way



Functional is Good

- No side effects
- Type safe
- Fine-grained control

But sometimes it's too much of a good thing ...

- Sea of parameters,
- most of which hardly ever change.
- Repetitive, boring, prone to mistakes

A more direct approach

If passing a lot of parameters gets tedious, leave some of them *implicit*.

A more direct approach

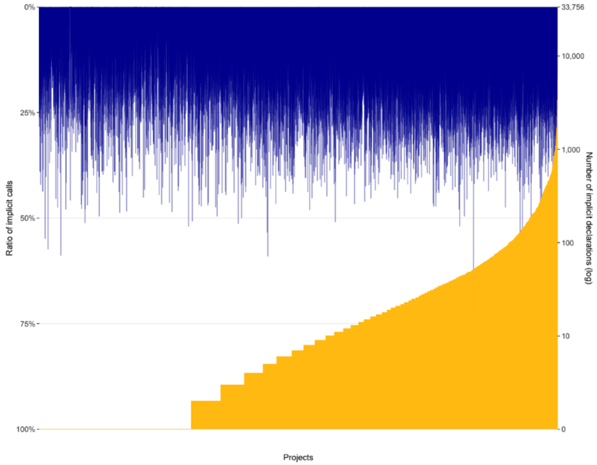
If passing a lot of parameters gets tedious, leave some of them *implicit*.

"Trade types for terms": We provide the type and have the compiler synthesize a matching term.

Ground Rules

- If you do not give an argument to an implicit parameter, one will be provided for you.
- Eligible are all implicit values that are visible at the point of call.
- If there are more than one eligible candidate, the most specific one is chosen.
- If there's no unique most specific candidate, an ambiguity error Is reported.

Implicits are Everywhere



At a glance:

- 7,280 Scala projects
- 18M lines of code
- 8M implicit call sites
- 370K implicit declarations

98% projects use implicits

78% of projects define implicits

27% of call sites use implicits

The top of the graph shows the ratio of call sites, in each project, that involves implicit resolution. The bottom shows the number of implicit definitions in each project (log scale).

Implement Type Classes

```
trait Ord[T] with
  def (x: T) less (y: T): Boolean

given intOrd: Ord[Int] with
  def (x: Int) less (y: Int) = x < y

given listOrd[T: Ord]: Ord[List[T]] with
  def (xs: List[T]) less (ys: List[T]) = (xs, ys) match
    case (_, Nil) => false
    case (Nil, _) => true
    case (x :: xs1, y :: ys1) => x < y || x == y && xs1.less(ys1)
```

Note: `[T: Ord]` is a *context bound*, it's a shorthand for an implicit parameter

Implement Type Classes

```
trait Ord[T] with
  def (x: T) less (y: T): Boolean

given intOrd: Ord[Int] with
  def (x: Int) less (y: Int) = x < y

given listOrd[T](given Ord[T]): Ord[List[T]] with
  def (xs: List[T]) less (ys: List[T]) = (xs, ys) match
    case (_, Nil) => false
    case (Nil, _) => true
    case (x :: xs1, y :: ys1) => x < y || x == y && xs1.less(ys1)
```

Note: `[T: Ord]` is a *context bound*, it's a shorthand for an implicit parameter

Implement Type Classes

```
trait Ord[T] with
  def (x: T) less (y: T): Boolean

given intOrd: Ord[Int] with
  def (x: Int) less (y: Int) = x < y

given listOrd[T: Ord]: Ord[List[T]] with
  def (xs: List[T]) less (ys: List[T]) = (xs, ys) match
    case (_, Nil) => false
    case (Nil, _) => true
    case (x :: xs1, y :: ys1) => x < y || x == y && xs1.less(ys1)
```

Note: The names `intOrd` and `listOrd` can also be left out, i.e. given instances can be anonymous.

Implement Type Classes

```
trait Ord[T] with
  def (x: T) less (y: T): Boolean

given Ord[Int] with
  def (x: Int) less (y: Int) = x < y

given [T: Ord]: Ord[List[T]] with
  def (xs: List[T]) less (ys: List[T]) = (xs, ys) match
    case (_, Nil) => false
    case (Nil, _) => true
    case (x :: xs1, y :: ys1) => x < y || x == y && xs1.less(ys1)
```

Note: The names `intOrd` and `listOrd` can also be left out, i.e. given instances can be anonymous.

Establish Context

Example: A conference management system.

Reviewers should only see (directly or indirectly) the scores of papers where they have no conflict with an author.

```
class Viewers(val persons: Set[Person])

def hasConflict(ps1: Set[Person], ps2: Set[Person]) =
  ps2.exists(ps1 contains _)
```

A Conference Management System

```
def viewers(given vs: Viewers): Set[Person] =
  vs.persons

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

def viewRankings(given Viewers): List[Paper] =
  papers.sortBy(score(_))

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

A Conference Management System

```scala
def viewers(given vs: Viewers): Set[Person] =
  vs.persons

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

def viewRankings(given Viewers): List[Paper] =
  papers.sortBy(score(_))

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

Context is usually stable, can change at specific points (e.g. in `delegateTo`).

Scrap the Boilerplate

Observation: It gets tedious to write all these given Viewers parameters!

Having to write `(given Viewers)` a couple of times does not look so bad.

But in the new Scala compiler `dotc` there are $>$ 2600 occurrences of the parameter `(given Context)`

Would it not be nice to get rid of them?

Implicit Function Types

Let's massage the definition of viewRankings a bit:

```
def viewRankings = (given Viewers) =>
  papers.sortBy(score(_))
```

What is its type?

Implicit Function Types

Let's massage the definition of viewRankings a bit:

```
def viewRankings = (given Viewers) =>
  papers.sortBy(score(_))
```

What is its type?

```
def viewRankings: (given Viewers) => List[Paper]
```

A function type that takes implicit parameters is caled an *implicit function type*.

Two Rules for Typing

1. Implicit functions get implicit arguments just like implicit methods. Given

   ```
   def f: (given A) => B
   given a: A
   ```

   the expression `f` expands to `f(a)`
2. Implicit functions get created on demand. If the expected type of `b` is `(given A) => B`, then `b` expands to

```
(given Viewers) => b
```

or, inserting an arbitrary compiler-generate name for the parameter

```
(given $ev: Viewers) => b
```

Using Implicit Function Types

Assume:

```
type Viewed[T] = (given Viewers) => T
```

Then reformulate:

```
def score(paper: Paper): Viewed[Int] =
  if hasConflict(viewers, paper.authors) then -100
  else realScore(paper)

def viewRankings: Viewed[List[Paper]] =
  papers.sortBy(score(_))

def delegateTo[T](query: Viewed[T], p: Person): Viewed[T] =
  query(given Viewers(viewers + p))
```

Trade Types for Parameters

- By specifying a type of an expression e, we can provide in a very general way a context for e.
- Types can be declared directly, or be inferred with local type inference.

Another Example: The Builder Pattern

Neat way to define structure-building DSLs, like this:

```
table {
  row {
    cell(”top left”)
    cell(”top right”)
  }
  row {
    cell(”botttom left”)
    cell(”bottom right”)
  }
}
```

Natively supported in Groovy and in Kotlin via “receiver functions”.

Scala Implementation: Classes

```
class Table with
  val rows = new ArrayBuffer[Row]
  def add(r: Row): Unit = rows.append(r)
  override def toString = rows.mkString(”Table(”, ”, ”, ”)”)

class Row with
  val cells = new ArrayBuffer[Cell]
  def add(c: Cell): Unit = cells.append(c)
  override def toString = cells.mkString(”Row(”, ”, ”, ”)”)
```

Scala Implementation: Constructors

```scala
def table(init: (given Table) => Unit) =
  val t = Table()
  init(given t)
  t

def row(init: (given Row) => Unit)(given t: Table) =
  val r = Row()
  init(given r)
  t.add(r)

def cell(str: String)(given r: Row) =
  r.add(Cell(str))
```

Givens in Action

```
table {
  row {
    cell(”top left”)
    cell(”top right”)
  }
  row {
    cell(”botttom left”)
    cell(”bottom right”)
  }
}
```

expands to:

Givens in Action

```
table { (given $t: Table) =>
  row { (given $r: Row) =>
    cell(”top left”)(given $r)
    cell(”top right”)(given $r)
  }(given $t)
  row { (given $r: Row) =>
    cell(”botttom left”)(given $r)
    cell(”bottom right”)(given $r)
  }(given $t)
}
```

Reference

Simplicitly -
Foundation and Applications of Implicit Function Types
Martin Odersky, Olivier Blanvillain, Fenyun Liu, Aggelos Biboudis, Heather Miller, Sandro Stucki
POPL 2018