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

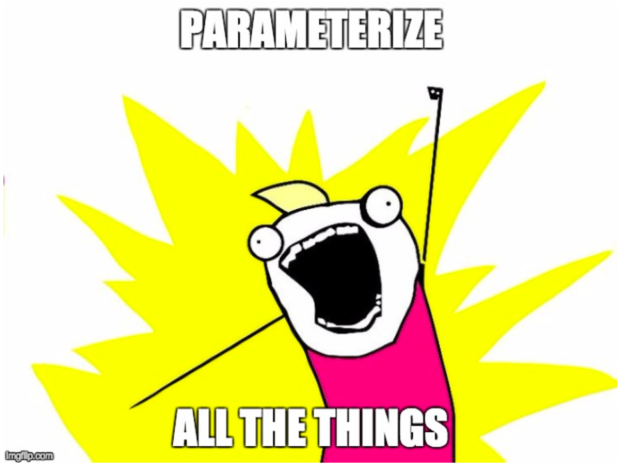
- rigid if immutable,
- unsafe if mutable

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

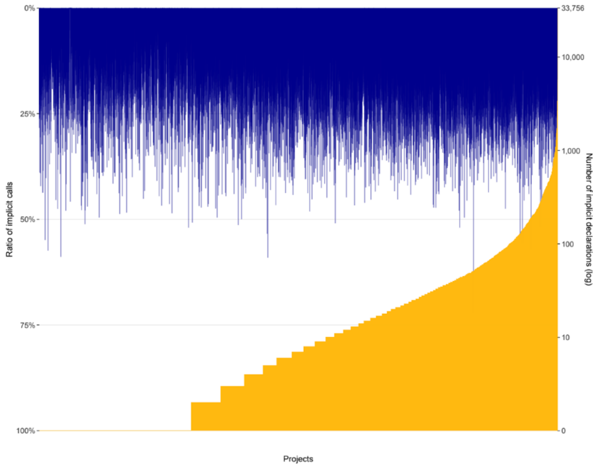
If passing a lot of parameters gets tedious, leave some of them *implicit*.

“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

```scala
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 += r
  override def toString = rows.mkString(”Table(”, ”, ”, ”)”)

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

Scala Implementation: Constructors

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

def row(init: (given Row) => Unit)(given t: Table) =
  given r: Row
  init
  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 $t: Table) =>
    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

Imperative Event Handling: The Observer Pattern

Principles of Functional Programming

Martin Odersky

The Observer Pattern

The Observer Pattern is widely used when views need to react to changes in a model.

Variants of it are also called

- publish/subscribe
- model/view/controller (MVC).

A Publisher Trait

```
trait Publisher with

  private var subscribers: Set[Subscriber] = Set()

  def subscribe(subscriber: Subscriber): Unit =
    subscribers += subscriber

  def unsubscribe(subscriber: Subscriber): Unit =
    subscribers -= subscriber

  def publish(): Unit =
    subscribers.foreach(_.handler(this))
```

A Subscriber Trait

```
trait Subscriber with
  def handler(pub: Publisher)
```

Observing Bank Accounts

Let's make BankAccount a Publisher:

```
class BankAccount extends Publisher with
  private var balance = 0

  def deposit(amount: Int): Unit =
    if amount > 0 then
      balance = balance + amount

  def withdraw(amount: Int): Unit =
    if 0 < amount && amount <= balance then
      balance = balance - amount

    else throw Error("insufficient funds")
```

Observing Bank Accounts

Let's make `BankAccount` a `Publisher`:

```
class BankAccount extends Publisher with
  private var balance = 0
  def currentBalance: Int = balance          // <---
  def deposit(amount: Int): Unit =
    if amount > 0 then
      balance = balance + amount
      publish()                              // <---
  def withdraw(amount: Int): Unit =
    if 0 < amount && amount <= balance then
      balance = balance - amount
      publish()                              // <---
    else throw Error(”insufficient funds”)
```

An Observer

A `Subscriber` to maintain the total balance of a list of accounts:

```
class Consolidator(observed: List[BankAccount]) extends Subscriber with
  observed.foreach(_.subscribe(this))

  private var total: Int = _
  compute()

  private def compute() =
    total = observed.map(_.currentBalance).sum

  def handler(pub: Publisher) = compute()

  def totalBalance = total
```

Observer Pattern, The Good

- Decouples views from state
- Allows to have a varying number of views of a given state
- Simple to set up

Observer Pattern, The Bad

- Forces imperative style, since handlers are `Unit`-typed
- Many moving parts that need to be co-ordinated
- Concurrency makes things more complicated
- Views are still tightly bound to one state; view update happens immediately.

To quantify (Adobe presentation from 2008):

- $1/3^{rd}$ of the code in Adobe's desktop applications is devoted to event handling.
- $1/2$ of the bugs are found in this code.

How to Improve?

During the rest of this session we will explore a different way, namely *functional reactive programming*, in which we can improve on the imperative view of reactive programming embodied in the observer pattern.

Functional Reactive Programming

Principles of Functional Programming

Martin Odersky

What is FRP?

Reactive programming is about reacting to sequences of *events* that happen in *time*.

Functional view: Aggregate an event sequence into a *signal*.

- A signal is a value that changes over time.
- It is represented as a function from time to the value domain.
- Instead of propagating updates to mutable state, we define new signals in terms of existing ones.

Example: Mouse Positions

Event-based view:

Whenever the mouse moves, an event

```
MouseMoved(toPos: Position)
```

is fired.

FRP view:

A signal,

```
mousePosition: Signal[Position]
```

which at any point in time represents the current mouse position.

Origins of FRP

FRP started in 1997 with the paper *Functional Reactive Animation* by Conal Elliott and Paul Hudak and the *Fran* library.

There have been many FRP systems since, both standalone languages and embedded libraries.

Some examples are: *Flapjax*, *Elm*, *React.js*

Event streaming dataflow programming systems such as Rx are related but the term FRP is not commonly used for them.

We will introduce FRP by means of of a minimal class, `frp.Signal` whose implementation is explained at the end of this module.

frp.Signal is modelled after Scala.react, which is described in the paper *Deprecating the Observer Pattern*.

Fundamental Signal Operations

There are two fundamental operations over signals:

1. Obtain the value of the signal at the current time.
In our library this is expressed by `()` application.

```
mousePosition()   // the current mouse position
```

Fundamental Signal Operations

There are two fundamental operations over signals:

1. Obtain the value of the signal at the current time.
In our library this is expressed by `()` application.

```
mousePosition()    // the current mouse position
```

2. Define a signal in terms of other signals.
In our library, this is expressed by the `Signal` constructor.

```
def inReactangle(LL: Position, UR: Position): Signal[Boolean] =
  Signal {
    val pos = mousePosition()
    LL <= pos && pos <= UR
  }
```

Constant Signals

The `Signal(...)` syntax can also be used to define a signal that has always the same value:

```
val sig = Signal(3)      // the signal that is always 3.
```

Time-Varying Signals

How do we define a signal that varies in time?

- We can use externally defined signals, such as `mousePosition` and map over them.
- Or we can use a `Signal.Var`.

Variable Signals

Values of type `Signal` are immutable.

But our library also defines a subclass `Signal.Var` of `Signal` for signals that can be changed.

`Var` provides an "update" operation, which allows to redefine the value of a signal from the current time on.

```
val sig = Signal.Var(3)
sig.update(5)                // From now on, sig returns 5 instead of 3.
```

Aside: Update Syntax

In Scala, calls to `update` can be written as assignments.

For instance, for an array `arr`

```
arr(i) = 0
```

is translated to

```
arr.update(i, 0)
```

which calls an `update` method which can be thought of as follows:

```
class Array[T] with
  def update(idx: Int, value: T): Unit
  ...
```

Aside: Update Syntax

Generally, an indexed assignment like $\texttt{f}(\texttt{E}_1, \ldots, \texttt{E}_n) = \texttt{E}$

is translated to $\texttt{f.update}(\texttt{E}_1, \ldots, \texttt{E}_n, \texttt{E})$.

This works also if $n = 0$: $\texttt{f}() = \texttt{E}$ is shorthand for $\texttt{f.update}(\texttt{E})$.

Hence,

```
sig.update(5)
```

can be abbreviated to

```
sig() = 5
```

Signals and Variables

Signals of type `Var` look a bit like mutable variables, where

```
sig()
```

is dereferencing, and

```
sig() = newValue
```

is update.

But there's a crucial difference:

We can apply functions to signals, which gives us a relation between two signals that is maintained automatically, at all future points in time.

No such mechanism exists for mutable variables; we have to propagate all updates manually.

Example

Repeat the `BankAccount` example of the last section with signals.

Add a signal `balance` to `BankAccounts`.

Define a function `consolidated` which produces the sum of all balances of a given list of accounts.

What savings were possible compared to the publish/subscribe implementation?

Signals and Variables (2)

Note that there's an important difference between the variable assignment

```
v = v + 1
```

and the signal update

```
s() = s() + 1
```

In the first case, the *new* value of `v` becomes the *old* value of `v` plus 1.

In the second case, we try define a signal `s` to be *at all points in time* one larger than itself.

This obviously makes no sense!

Exercise

Consider the two code fragments below

1.
   ```
   val num = Signal(1)
   val twice = Signal(num() * 2)
   num() = 2
   ```

2.
   ```
   var num = Signal(1)
   val twice = Signal(num() * 2)
   num = Signal(2)
   ```

Do they yield the same final value for `twice()`?

O yes
O no

Exercise

Consider the two code fragments below

1.
   ```
   val num = Signal(1)
   val twice = Signal(num() * 2)
   num() = 2
   ```

2.
   ```
   var num = Signal(1)
   val twice = Signal(num() * 2)
   num = Signal(2)
   ```

Do they yield the same final value for `twice()`?

```
O   yes
X   no
```

A Simple FRP Implementation

Principles of Functional Programming

Martin Odersky

A Simple FRP Implementation

We now develop a simple implementation of `Signal`s and `Var`s, which together make up the basis of our approach to functional reactive programming.

The classes are assumed to be in a package `frp`.

Their user-facing APIs are summarized in the next slides.

Summary: The Signal and Var APIs

```
trait Signal[T+] with
  def apply(): T = ???

object Signal with
  def apply[T](expr: => T) = ???

  class Var[T](expr: => T) extends Signal[T]
    def update(expr: => T): Unit = ???
```

Note that `Signals` are covariant, but `Vars` are not.

Implementation Idea

Each signal maintains

- its current value,
- the current expression that defines the signal value,
- a set of *observers*: the other signals that depend on its value.

Then, if the signal changes, all observers need to be re-evaluated.

Dependency Maintenance

How do we record dependencies in `observers`?

- When evaluating a signal-valued expression, need to know which signal `caller` gets defined or updated by the expression.
- If we know that, then executing a `sig()` means adding `caller` to the `observers` of `sig`.
- When signal `sig`'s value changes, all previously observing signals are re-evaluated and the set `sig.observers` is cleared.
- Re-evaluation will re-enter a calling signal `caller` in `sig.observers`, as long as `caller`'s value still depends on `sig`.
- For the moment, let's assume that `caller` is provided "magically"; we will discuss later how to make that happen.

Implementation of Signals

The `Signal` trait is implemented by a class `AbstractSignal` in the `Signal` object. This is a useful and common implementation technique that allows us to hide global implementation details in the enclosing object.

```
type Caller = AbstractSignal[?]

abstract class AbstractSignal[+T] extends Signal[T] with
  private var currentValue: T = _
  private var observers: Set[Caller] = Set()

  def apply(): T =
    observers += caller
    currentValue

  protected def eval: () => T   // evaluate the signal
```

(Re-)Eevaluating Signal Values

A signal value is evaluated using `computeValue()`

- on initialization,
- when an observed signal changes its value.

Here is its implementation:

```
protected def computeValue(): Unit =
  val newValue = eval()
  val observeChange = observers.nonEmpty && newValue != currentValue
  currentValue = newValue
  if observeChange then
    val obs = observers
    observers = Set()
    obs.foreach(_.computeValue())
```

Creating Signals

Signals are created with an `apply` method in the `Signal` object

```
object Signal with
  def apply[T](expr: => T): Signal[T] =
    new AbstractSignal[T] with
      val eval = () => expr
      computeValue()
```

Signal Variables

The `Signal` object also defines a class for variable signals with an `update` method

```
class Var[T](initExpr: => T) extends AbstractSignal[T] with
  protected var eval = () => initExpr
  computeValue()

  def update(newExpr: => T): Unit =
    eval = () => newExpr
    computeValue()
end Var
```

Who’s Calling?

How do we find out on whose behalf a signal expression is evaluated?

The most robust way of solving this is to pass the caller along to every expression that is evaluated.

So instead of having a by-name parameter

```
    expr: => T
```

we’d have a function

```
    expr: Caller => T
```

and when evaluating a signal, `s()` becomes `s(caller)`

Problem: This causes a lot of boilerplate code, and it’s easy to get wrong!

Calling, Implicitly

How about we make signal evaluation expressions implicit function types?

```
expr: (given Caller) => T
```

Then all caller parameters are passed implicitly.

New Signal and Var APIs

```
trait Signal[T+] with
  def apply(given caller: Signal.Caller): T

object Signal with
  def apply[T](expr: (given Caller) => T): Signal[T] = ???

  class Var[T](expr: (given Caller) => T) extends AbstractSignal[T]
    def update(expr: => T): Unit = ???
```

Summary

We have given a quick tour of functional reactive programming, with some usage examples and an implementation.

This is just a taster, there's much more to be discovered.

In particular, we only covered one particular style of FRP: Discrete signals changed by events.

Some variants of FRP also treat continuous signals.

Values in these systems are often computed by sampling instead of event propagation.