

PARTIAL FUNCTIONS IN SCALA

`https://github.com/logicalguess/partial-functions`

April 2017

OUTLINE

INTRODUCTION

- Motivation

- The PartialFunction Type

- Important PartialFunction[A, B] Methods

LOOPING AND BRANCHING

- Looping with Partial Functions

- Branching with Partial Functions

PROGRAM LOGIC AS COMPUTATIONAL GRAPHS

- Direct Composition of (Partial) Functions

- Conversion of (Partial) Functions to Flows

FROM METHODS TO EVENT-DRIVEN FUNCTIONS

- Class Based Objects

- Identifying The Events and State Types

- Writing The Logic as a Partial Function

- Wrapping Logic Into Entities

MOTIVATION

- ▶ Functions associate a **single** output to **each** input
- ▶ In Scala inputs and outputs are types: `Function[I, O]`
- ▶ There are many ways to create associations (e.g. formulas, tuples)
- ▶ Sometimes it is not possible or desired to associate an output to each input
- ▶ Division by zero is an example of a computation that needs to **skip** some inputs
- ▶ `Map[K, V]` is another prototypical example (many keys are **skipped**)

THE PARTIALFUNCTION TYPE

KEY FACTS

- ▶ `PartialFunction[-I, +O] <: Function[-I, +O]`
- ▶ `isDefinedAt(input:I): Boolean`
- ▶ `apply(input: I): O`

INTERESTING FACTS

- ▶ `List` and `Map` extend `PartialFunction`
- ▶ `Traversable`'s `collect` method takes a `PartialFunction` as an argument to process and skip elements at the same time
- ▶ `Sources` and `Flows` in `Akka` have the `collect` method too

IMPORTANT PARTIALFUNCTION[A, B] METHODS

METHODS/OPERATORS

- ▶ `applyOrElse[A1 <: A, B1 >: B](x: A1, default: A1 => B1): B1`
- ▶ `andThen[C](k: B => C): PartialFunction[A, C]`
- ▶ `orElse[A1 <: A, B1 >: B](that: PartialFunction[A1, B1]): PartialFunction[A1, B1]`
- ▶ `lift: A => Option[B]`

LOOPING WITH PARTIAL FUNCTIONS

PartialFunctions have a natural body of executable code (the apply method) that can be called repeatedly, and a natural termination condition (when isDefinedAt returns false)

USAGE EXAMPLE

```
val fibs: PartialFunction[(Int, Int), (Int, Int)] = {  
  case (f1: Int, f2: Int) if (f1 >= 0) => (f1 + f2, f1)  
}  
  
val restrict: PartialFunction[(Int, Int), (Int, Int)] = {  
  case (f1: Int, f2: Int) if f1 < 15 => (f1, f2)  
}
```

```
Iterator(restrict andThen fibs)((1, 1)) should be (21, 13)
```

BRANCHING WITH PARTIAL FUNCTIONS

PartialFunctions can be combined, but unfortunately `orElse` is not very useful in that respect since it restricts the domain of the resulting function to the intersection type of the two functions.

USAGE EXAMPLE

```
val int: PartialFunction[Int, Int] = {  
  case i: Int if i >= 5 => i  
}  
  
val string: PartialFunction[String, Int] = {  
  case s: String => s.length  
}  
  
val bool: PartialFunction[Boolean, Int] = {  
  case true => 1;  
  case false => 0  
}  
  
val nothing: PartialFunction[Any, Nothing] = {  
  case _ => throw new RuntimeException  
}
```

```
val branch = Branch(int, string)  
  
branch(77) shouldBe 77  
branch("xyz") shouldBe 3  
a [MatchError] should be thrownBy branch(true)
```

```
val branch = TypedBranch(int, string)  
  
branch(77) shouldBe 77  
branch("xyz") shouldBe 3  
//branch(true) //compile time error
```

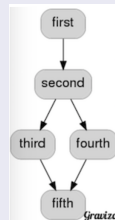
DIRECT COMPOSITION OF (PARTIAL) FUNCTIONS

Programs consist of computation steps represented by variable manipulation, expressions, functions/method invocations, flow control, etc.

The discipline of breaking code logic into functions (including high-order ones) that can be composed and reused is (together with immutable state) the essence of functional programming.

EXAMPLE

```
val convert: PartialFunction[Int, String] = {  
  case i: Int if i > 0 => i.toString  
}  
  
val bang: Function[String, String] = { s: String => s + "!" }  
val hash: Function[String, String] = { s: String => s + "##" }  
val concatenate: Function[(String, String), String] = { s => s._1 + s._2 }  
  
val diamondComposition: Int => String = i => {  
  val s = convert(i)  
  val b = bang(s)  
  val h = hash(s)  
  concatenate(b, h)  
}
```



CONVERSION OF (PARTIAL) FUNCTIONS TO FLOWS

PartialFunctions can be (implicitly) converted to Akka flows, so the full power of the stream DSL can be used to build computational graphs using existing functions.

More generally, the conversion/lifting of functions to any Monad will enable out-of-the-box composition (e.g. using for comprehensions).

EXAMPLE

```
val diamondFlow: Flow[Int, String, _] = Flow.fromGraph(GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._

  val input = builder.add(Flow[Int])
  val bcast = builder.add(Broadcast[String](2))
  val zip = builder.add(Zip[String, String])
  val output = builder.add(Flow[String])

  input.out ~> convert ~> bcast ~> bang ~> zip.in0
  bcast ~> hash ~> zip.in1
  zip.out ~> concatenate ~> output

  FlowShape(input.in, output.out)
})
```

CLASS BASED OBJECTS

- ▶ Typical object-oriented programming relies on class-based languages that allow defining state and behavior logic into instance prototypes.
- ▶ The names, the inputs and outputs of the methods as well as the state management of objects are free-form and usually poorly addressed.
- ▶ Design patterns have helped to some extent, but generally speaking developers cherish this freedom.

```
class Echo {  
  private var volume: Int = 5  
  private var playing: Option[String] = None  
  
  //volume logic  
  def volumeUp(): Unit = volume += 1  
  def volumeDown(): Unit = volume -= 1  
  def setVolume(v: Int): Unit = volume = v  
  def getVolume: String = s"The current volume is $volume"  
  
  //music logic  
  def play(piece: String): Unit = {  
    | playing = Some(piece)  
  }  
  def stop(): Unit = playing = None  
  
  //temperature logic  
  def getTemperature(place: String): Option[String] = {  
    | val temp = 70 //call service  
    | Some(s"The current temperature in $place is $temp degrees")  
  }  
}
```

IDENTIFYING THE EVENTS AND STATE TYPES

Method names and parameter types can be represented by case classes or objects in Scala.

THE TYPES

```
// events
sealed trait Event

object VolumeUp extends Event
object VolumeDown extends Event
case class SetVolume(level: Int) extends Event
object GetVolume extends Event

case class Play(piece: String) extends Event
object Stop extends Event

case class GetTemperature(place: String) extends Event

//state
case class State(volume: Int = 5,
                 playing: Option[String] = None,
                 response: Option[String] = None)
```

WRITING THE LOGIC AS A PARTIAL FUNCTION

The values return from methods are views into the state, which represents the "value" of an object instance.

THE LOGIC

```
//function
val logic: PartialFunction[(Event, State), State] = {
  case (VolumeUp, s) => s.copy(s.volume + 1) //validate
  case (VolumeDown, s) => s.copy(s.volume - 1) //validate
  case (SetVolume(l), s) => s.copy(l) //validate

  case (GetVolume, s) => s.copy(s.volume, s.playing,
    Some(s"The current volume is ${s.volume}")) //validate

  case (Play(p), s) => s.copy(s.volume, Some(p))
  case (Stop, s) =>
    if (s.playing.isDefined) s.copy(s.volume, None) else s

  case (GetTemperature(place), s) => {
    val temp = 70 //call service
    s.copy(s.volume, s.playing,
      Some(s"The current temperature in $place is $temp degrees"))
  }
}
```

WRAPPING LOGIC INTO ENTITIES

The types and logic can be reused and wrapped into Flows, Actors or more functional classes.

KEEPING THE LOGIC SEPARATE FROM BOILERPLATE

```
case class EchoScan(sink: Sink[State, _])
    (implicit mat: ActorMaterializer) {

    val source: Source[Event, SourceQueueWithComplete[Event]] =
        Source.queue[Event](10, OverflowStrategy.backpressure)

    val computation: SourceQueueWithComplete[Event] =
        source.via(logic).to(sink).run()

    def receive(event: Event): Unit = {
        computation.offer(event)
    }
}
```

```
class EchoActor extends Actor with LazyLogging {
    private var state: State = State()

    override def receive: Receive = {
        case msg: Event => {
            state = logic(msg, state)
            sender() ! state
        }
    }
}
```

```
class FunctionalEcho {
    private var state: State = State()

    def receive(event: Event): State = {
        state = logic(event, state) //could throw state
    }

    def getState: State = state
}
```