Chapter 3: The Logic of Types, Part II

Disjunction types

Sergei Winitzki

Academy by the Bay

December 8, 2017

Tuples with names: "case classes"

- Pair of values: `val a: (Int, String) = (123, "xyz")`
- For convenience, we can define a name for this type:
  `type MyPair = (Int, String); val a: MyPair = (123, "xyz")`
- We can define a name for each value and also for the type:
  ```
  case class MySocks(size: Double, color: String)
  val a: MySocks = MySocks(10.5, "white")
  ```
- Case classes can be nested:
  ```
  case class BagOfSocks(socks: MySocks, count: Int)
  val bag = BagOfSocks(MySocks(10.5, "white"), 6)
  ```
- Parts of the case class can be accessed by name:
  `val c: String = bag.socks.color`
- Parts can be given in any order by using names:
  `val y = MySocks(color = "black", size = 11.0)`
- Default values can be defined for parts:
  ```
  case class Shirt(color: String = "blue", hasHoles: Boolean = false)
  val sock = Shirt(hasHoles = true)
  ```

Tuples with one element and with zero elements

- A tuple type expression `(Int, String)` is special syntax for parameterized type `Tuple2[Int, String]`
- Case class with no parts is called a "case object"
- What are tuples with one element or with zero elements?
  - There is no `Tuple0` – it is called `Unit`

Tuples	Case classes
`(123, "xyz"): Tuple2[Int, String]`	`case class A(x: Int, y: String)`
`(123,): Tuple1[Int]`	`case class B(z: Int)`
`(): Unit`	`case object C`

- Case classes can have one or more type parameters:
  `case class Pairs[A, B](left: A, right: B, count: Int)`
- The "`Tuple`" types could be defined by this code:
  `case class Tuple2[A, B](_1: A, _2: B)`

Pattern-matching syntax for case classes

Scala allows pattern matching in two places:

- `val` *pattern* `= ...` (value assignment)
- `case` *pattern* `⇒ ...` (partial function)

Examples with case classes:

- ```
  val a = MySocks(10.5, "white")
  val MySocks(x, y) = a
  ```
- `val f: BagOfSocks⇒Int = { case BagOfSocks(MySocks(s, c), z)⇒...}`
- ```
  def f(b: BagOfSocks): String = b match {
    case BagOfSocks(MySocks(s, c), z) ⇒ c
  }
  ```
- Note: `s`, `c`, `z` are defined as **pattern variables** of correct types

Disjunction types

- Motivational examples:
  - The roots of a quadratic equation can be either a pair, or one, or none
  - Binary search gives either a found value and an index, or nothing
  - Computations that give a value or an error with a text message
  - Computer game states: several kinds of rooms, types players, etc.
    - ⋆ Each kind of room may have different sets of properties
- We would like to be able to represent *disjunctions* of domains
  - A value that is *either* `(Complex, Complex)` or `Complex` or empty `()`
  - A value that is *either* `(Int, Int)` or empty `()`
  - A value that is *either* an `Int` value or a `String` error message
  - A value that is one case class out of a number of case classes
- Disjunction types represent such values as types

Disjunction types: mathematical point of view

- The type of the function's argument represents the function's *domain*
  - For example: $f(x)$ where $x \in \mathbb{R}$
- We would like to be able to represent arbitrary *disjoint* domains
  - For example: x is either a point on a line or a point on a surface
- In functional programming, the disjoint domains are always **labeled**
  - For example: $x \in (\text{left}, \mathbb{R}) \cup \left(\text{right}, \mathbb{R}^2\right)$
    - ★ The disjoint union is always an *exclusive-or*
    - ★ Labels come from a fixed, special set of symbols
  - Given any such x, we can determine its "side" of the union
  - We can obtain the corresponding value in each case
- In Scala, this type is denoted `Either[Double, (Double, Double)]`
- Pattern-matching is used to define expressions with such types

Disjunction type: `Either[A, B]`

Example: `Either[String, Int]` (may be used for error reporting)

- Represents a value that is *either* a `String` or an `Int` (but not both)
- Example values: `Left("blah")` or `Right(123)`
- Use pattern matching to distinguish "left" from "right":

```
def logError(x: Either[String, Int]): Int = x match {
  case Left(error) ⇒ println(s"Got error: $error"); -1
  case Right(res) ⇒ res
} // Left("blah") and Right(123) are possible values of type Either[String, Int]
```

- Now `logError(Right(123))` returns `123` while `logError(Left("bad result"))` prints the error and returns `-1`
- The `case` expression chooses among possible values of a given type
  - Note the similarity with this code:

```
def f(x: Int): Int = x match {
  case 0 ⇒ println(s"error: must be nonzero"); -1
  case 1 ⇒ println(s"error: must be greater than 1"); -1
  case res ⇒ res
} // 0 and 1 are possible values of type Int
```

More general disjunction types: trait + case classes

A future version of Scala 3 has a short syntax for disjunction types:

- `type MyIntOrStr = Int | String`
- more generally, `type MyType = List[Int] | (Int, Boolean) | MySocks`
    - Some libraries (scalaz, cats, shapeless) also provide shorter syntax

For now, in Scala 2, we use the "long syntax":
(specify names for each case and for each part, use "`trait`" / "`extends`")

```
sealed trait MyType
final case class HaveListInt(x: List[Int]) extends MyType
final case class HaveIntBool(s: Int, b: Boolean) extends MyType
final case class HaveSocks(socks: MySocks) extends MyType
```

Pattern-matching example:

```
val x: MyType = if (...) HaveSocks(...) else HaveListInt(...)
... // some other code here
x match {
  case HaveListInt(lst) ⇒ ...
  case HaveIntBool(p, q) ⇒ ...
  case HaveSocks(s) ⇒ ...
}
```

The most often used disjunction type: Option[T]

A simple implementation:

```
sealed trait Option[T]
final case class Some[T](t: T) extends Option[T]
final case object None extends Option[Nothing]
```

Pattern-matching example:

```
def saveDivide(x: Double, y: Double): Option[Double] = {
  if (y == 0) None else Some(x / y)
// Example usage:
val result = safeDivide(1.0, q) match {
  case Some(x) ⇒ previousResult * x
  case None ⇒ previousResult // provide a default value
}
```

Many Scala library functions return an `Option[T]`

- `find`, `headOption`, `reduceOption`, `get` (for `Map[K, V]`), `lift` for `Array[T]`
  - Note: `Option[T]` is "collection-like": has `map`, `flatMap`, `filter`, `exists`...

Worked examples

- What problems can we solve now?
  - Represent values from disjoint domains as a single type
  - Use such values in collections safely
- Define a disjunction type `DayOfWeek` representing the seven days.
- Modify `DayOfWeek` so that the values additionally represent a restaurant name and total amount for Fridays and a wake-up time on Saturdays.
- Define a disjunction type `RootsOfQuadratic` that represents real-valued roots of the equation $x^2 + bx + c = 0$ for arbitrary real b, c. (The cases of interest are: no real roots; two equal roots; two unequal roots.) Implement `solve2: ((Double, Double)) ⇒ RootsOfQuadratic`.
- Define a function `rootAverage: RootsOfQuadratic ⇒ Option[Double]` that computes the average value of all real roots, returning `None` if the average is undefined.
- Generate 100 random coefficients b, c (uniformly distributed between -1 and 1) and compute the mean of `rootAverage` for them.
- Implement `def f[A, B]: ((Option[A], Option[B])) ⇒ Option[(A, B)]`

Exercises II

1. Define a disjunction type `CellState` representing the visual state of one cell in the "Minesweeper" game: A cell can be either closed, or display a bomb, or be open and display the number of neighbor bombs.
2. Define a function from `Seq[Seq[CellState]]` to `Int`, counting the total number of cells with 0 neighbor bombs shown.
3. Define a disjunction type `RootOfLinear` representing all possibilities for the solution of the equation $ax + b = 0$ for arbitrary real a, b. (The cases of interest are: no roots; one root; all x are roots.) Implement the solution as a function `solve1: ((Double, Double)) ⇒ RootOfLinear`.
4. Given a `Seq[(Double, Double)]` containing pairs (a, b) of the coefficients of $ax + b = 0$, use `solve1` to produce a `Seq[Double]` containing the roots of that equation when a unique root exists.
5. Define fully parametric functions having these types:
   1. `def f1[A, B]: Option[(A, B)] ⇒ (Option[A], Option[B])`
   2. `def f2[A, B]: Either[A,B] ⇒ (Option[A], Option[B])`
   3. `def f3[A,B,C]: Either[A, Either[B,C]] ⇒ Either[Either[A,B], C]`