Chapter 3: The Logic of Types

Sergei Winitzki

Academy by the Bay

November 22, 2017

Tuples with names, or “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 a special type 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 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: using 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`

For now, in Scala 2, we use the "long syntax":

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

Pattern-matching example:

```
val x: MyType = ???
x match {
  case HaveListInt(lst) => ...
  case HaveIntBool(p, q) => ...
  case HaveSocks(s) => ...
}
```

Types and propositional logic

The Curry-Howard correspondence

This code: `val x: SomeTypeBlah = ...` means that we can compute a value of type `SomeTypeBlah` as part of our program

- Let's denote this *proposition* by $\mathcal{CH}(T)$ – "Code Has a value of type `T`"
- We have the following correspondence:

Type	Proposition	Short notation
`T`	$\mathcal{CH}(T)$	T
`(A, B)`	$\mathcal{CH}(A)$ *and* $\mathcal{CH}(B)$	$A\,\&\,B$
`Either[A, B]`	$\mathcal{CH}(A)$ *or* $\mathcal{CH}(B)$	$A \mid B$
`A ⇒ B`	$\mathcal{CH}(A)$ *implies* $\mathcal{CH}(B)$	$A \Rightarrow B$
`Unit`	*true*	1
`Nothing`	*false*	0

- type parameter `[T]` means $\forall T$, for example the type of the function `def dupl[A](x: A): (A, A)` corresponds to the (valid) proposition: $\forall A : \mathcal{CH}(A)$ *implies* $(\mathcal{CH}(A)$ *and* $\mathcal{CH}(A))$

Working with the CH correspondence

- What problems can we solve now?

Working with the CH correspondence

Implications for programming language design

- The CH correspondence maps the type system of each programming language into a certain system of logical propositions
- Scala, Haskell, OCaml, F#, Swift, Rust, etc. are mapped into the full constructive logic (all logical operations are available)
  - C, C++, Java, C#, etc. are mapped to *incomplete* logics – without "or" and without "true"
  - Python, JavaScript, Ruby, Clojure, etc. have only one type ("any value") and are mapped to logics with only one proposition
- The CH correspondence is a principle for designing type systems:
  - Choose a complete logic, free of inconsistency
    - Mathematicians have studied all kinds of logics and determined which ones are interesting, and found the minimal sets of axioms for them
  - Provide a type constructor for each basic operation (e.g. "*or*", "*and*")

Working with the CH correspondence

- What problems can we solve now?

Summary

- What problems can we solve now?

Exercises

1. Define a function of type `Seq[Double] => Seq[Double]` that "normalizes" the sequence: it finds the element having the max. absolute value and, if that value is nonzero, divides all elements by that factor.
2. Define a function of type `Seq[Seq[Int]] => Seq[Seq[Int]]` that adds 20 to every element of every inner sequence.
3. An integer n is called "3-factor" if it is divisible by only three different integers j such that $2 \leq j < n$. Compute the set of all "3-factor" integers n among $n \in [1, ..., 1000]$.
4. Given a function f of type `Int => Boolean`, an integer n is called "3-f" if there are only three different integers $j \in [1, ..., n]$ such that $f(j)$ returns `true`. Define a function that takes f as an argument and returns a sequence of all "3-f" integers among $n \in [1, ..., 1000]$. What is the type of that function? Rewrite Exercise 3 using that function.
5. Define a function that takes two functions `f: Int => Double` and `g: Double => String` as arguments, and returns a new function that computes the functional composition of `f` and `g`.