Chapter 3: The Logic of Types

Sergei Winitzki

Academy by the Bay

November 22, 2017

Types and syntax of functions returning functions

"Curried functions" in Scala

- A function that returns a function:

```
def logWith(topic: String): (String ⇒ Unit) = {
   x ⇒ println(s"$topic: $x")
}
```

- Calling this function:

```
val statusLogger: (String ⇒ Unit) = logWith("Result status")

primaryLogger("success")
```

- One-line syntax for calling: `logWith("Result status")("success")`
- Alternative syntax ("Curried" function):

```
val logWith: String ⇒ String ⇒ Unit =
  topic ⇒ x ⇒ println(s"$topic: $x")
```

- Syntax convention: `x ⇒ y ⇒ z` means `x ⇒ (y ⇒ z)`

Functions with fully parametric types

"No argument type left non-parametric"

Compare these two functions (note tuple type syntax):

```
def hypothenuse = (x: Double, y: Double) ⇒ math.sqrt(x*x + y*y)
def swap: ((Double, Double)) ⇒ (Double, Double) =
  { case (x, y) ⇒ (y, x) }
```

- We can fully parameterize the argument types for `swap`:

```
def swap[X, Y]: ((X, Y)) ⇒ (Y, X) = { case (x, y) ⇒ (y, x) }
```

- (The first function is too specific to generalize the argument types.)
- Note: Scala does not support a `val` with a parametric type
  - Instead we can use `def` or parametric classes/traits
- More examples:

```
def identity[T]: (T ⇒ T) = x ⇒ x
def const[C, X]: (C ⇒ X ⇒ C) = c ⇒ x ⇒ c
def compose[X, Y, Z](f: X ⇒ Y, g: Y ⇒ Z): X ⇒ Z = x ⇒ g(f(x))
```

- Functions with fully parametric types *are* actually useful!

Worked examples

- For the functions `const` and `identity` defined above, what is `const(identity)` and what is its type? Write out the type parameters.
- Define a function `twice` that takes a function f as its argument and returns a *function* that applies f twice. For example, `twice(x ⇒ x+3)` should return a function equivalent to `x ⇒ x+6`. Find the type of `twice`.
- What does `twice(twice)` do? Test your answer on the expression `twice(twice[Int])(x ⇒ x+3)(10)`. What are the type parameters here?
- Implement a function that applies a given function f repeatedly to an initial value x_0, until a given condition function `cond` returns true:

```
def converge[X](f: X ⇒ X, x0: X, cond: X ⇒ Boolean): X = ???
```

- Take a function with two arguments, fix the value of the first argument, and return the function of the remaining one argument. Define this operation as a function with fully parametric types:

```
def firstArg[X, Y, Z](f: (X, Y) ⇒ Z, x0: X): Y ⇒ Z = ???
```

- Infer missing types: `def p[...]:...  = f ⇒ f(2)`. Does `f(f)` compile?
- Infer missing types: `def p[...]:...  = f ⇒ g ⇒ g(f)`

Exercises I

- For the function `identity` defined above, what is `identity(identity)` and what is its type? Same question for `identity(const)`.
- For the function `const` above, what is `const(const)`, what is its type?
- For the function `twice` above, what does `twice(twice(twice)))` do? Test your answer on an example.
- Define a function `thrice` that applies its argument function 3 times, similarly to `twice`. What does `thrice(thrice(thrice)))` do?
- Define a function `ence` that applies a given function *n* times.
- Take a function with two arguments, and define a function of these two arguments swapped. Package this functionality as a function `swapFunc` with fully parametric types. To test:

```
def f(x: Int, y: Int) = x - y // check that f(10, 2) gives 8
val g = swapFunc(f)  // now check that g(10, 2) gives – 8
```

- Infer missing types: `def r[...]:...  = f ⇒ f(g ⇒ g(f))`
- Infer missing types: `def s[...]:...  = f ⇒ g ⇒ g(x ⇒ f(g(x)))`

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 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 types

- Motivational examples:
  - The roots of a quadratic equation are 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 sets
  - 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 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`
  - 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 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]`), etc.
  - Note: `Option[T]` is "collection-like": has `map`, `flatMap`, `filter`, `exists`...

Worked examples

- What problems can we solve now?

Exercises II

1. a

Types and propositional logic

The Curry-Howard correspondence

This code: `val x: T = ...` means that *we can compute a value* of type `T` 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* $\mathcal{CH}(B)$	$A \times B$
`Either[A, B]`	$\mathcal{CH}(A)$ *or* $\mathcal{CH}(B)$	$A \oplus B$
`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: $\forall A : A \Rightarrow A \times A$

Working with the CH correspondence

- Example 1:

```
sealed trait UserAction
case class SetName(first: String, last: String) extends UserAction
case class SetEmail(email: String) extends UserAction
case class SetUserId(id: Long) extends UserAction
```

- Short notation: $\texttt{UserAction} = (\texttt{String} \times \texttt{String}) \oplus \texttt{String} \oplus \texttt{Long}$
- Example 2: parametric type

```
sealed trait Either3[A, B, C]
case class Left[A, B, C](x: A) extends Either3[A, B, C]
case class Middle[A, B, C](x: B) extends Either3[A, B, C]
case class Right[A, B, C](x: C) extends Either3[A, B, C]
```

- Short notation: $\forall A \forall B \forall C : \text{Either3}[A, B, C] = A \oplus B \oplus C$

- Some standard identities in logic:

$$
\begin{aligned}
A \times 1 &= A \\
A + 1 &= 1 \\
(A \times B) \times C &= A \times (B \times C) \\
(A \oplus B) \oplus C &= A \oplus (B \oplus C) \\
A \times (B \oplus C) &= (A \times B) \oplus (A \times C) \\
A \oplus (B \times C) &= (A \oplus B) \times (A \oplus C)
\end{aligned}
$$

- Each identity gives functions that map both ways
- Some of these identities yield *isomorphisms of types*
  - Which ones do *not* yield isomorphisms, and why?

- Example 3: Recursive type

```
sealed trait IntList
final case object Empty extends IntList
final case class Nonempty(head: Int, tail: IntList) extends IntList
```

- Short notation: (the sign "$\equiv$" means type isomorphism)

$$\begin{aligned}\texttt{IntList} &\equiv 1 \oplus \texttt{Int} \times \texttt{IntList} \equiv 1 \oplus \texttt{Int} \times (1 \oplus \texttt{Int} \times (1 \oplus \texttt{Int} \times (...)...) \\ &\equiv 1 \oplus \texttt{Int} \oplus \texttt{Int} \times \texttt{Int} \oplus \texttt{Int} \times \texttt{Int} \times \texttt{Int} \oplus ...\end{aligned}$$

- Recursive list of integers $\equiv$ disjunction of empty, list of 1 integer, list of 2 integers, etc.

Working with the CH correspondence

- Any valid proposition can be implemented in code

Proposition	Code
$\forall A : A \Rightarrow A$	`def identity[A](x:A):A = x`
$\forall A : A \Rightarrow 1$	`def toUnit[A](x:A): Unit = ()`
$\forall A \forall B : A \Rightarrow A \oplus B$	`def inLeft[A,B](x:A): Either[A,B] = Left(x)`
$\forall A \forall B : A \times B \Rightarrow A$	`def first[A,B](p:(A,B)):A = p._1`
$\forall A \forall B : A \Rightarrow (B \Rightarrow A)$	`def const[A,B](x:A):B⇒A = (y:B)⇒x`

- Invalid propositions *cannot be implemented* in code
  - Examples:
    $\forall A : 1 \Rightarrow A$; $\forall A \forall B : A \oplus B \Rightarrow A$;
    $\forall A \forall B : A \Rightarrow A \times B$; $\forall A \forall B : (A \Rightarrow B) \Rightarrow A$
- Given a type, can we decide whether it is implementable?
  - Example: $\forall A \forall B : ((((A \Rightarrow B) \Rightarrow B) \Rightarrow A) \Rightarrow B) \Rightarrow B$
    - ⋆ Pure propositional logic has a decision algorithm

Worked examples

- What problems can we solve now?

Exercises III

1. a

Working with the CH correspondence

Implications for designing new programming languages

- 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" / "false"
  - 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

Implications for actually writing code

What problems can we solve now?

- Given a fully parametric type, decide whether it can be implemented in code ("type is inhabited"); if so, *generate* the code
  - The Gentzen-Vorobiev-Hudelmaier algorithms and generalizations
- Given some code, infer the most general type it can have
  - The Damas-Hindley-Milner algorithm (Scala code) and generalizations