Some functional programming in Scala (and more)

Jacques Noyé
27 may 2013

1 Exercises

1.1 First order FP

Translate in Scala, as methods and as functions, the functions defined as follows in Haskell. Instead of *myAppend*, use the predefined concatenation operator :::

\[
\begin{align*}
\text{myLength} & : [a] \rightarrow \text{Int} \\
\text{myLength} & \ [\] = 0 \\
\text{myLength} & \ (_:xs) = 1 + \text{myLength} \ xs \\
\text{myReverse} & : [a] \rightarrow [a] \\
\text{myReverse} & \ (x:xs) = \text{myAppend} \ (\text{myReverse} \ xs) \ [x] \\
\text{myReverse} & \ xs = xs \\
\text{myReverse'} & : [a] \rightarrow [a] \\
\text{myReverse'} & \ xs = \text{myReverse''} \ xs \ [\] \\
& \quad \text{where} \ 
\text{myReverse''} & : [a] \rightarrow [a] \rightarrow [a] \\
& \quad \text{myReverse''} \ [\] \ a = a \\
& \quad \text{myReverse''} \ (x:xs) \ a = \text{myReverse''} \ xs \ (x:a)
\end{align*}
\]

You don’t need to curry your definitions, just use a single list of parameters. You can define methods either directly in the interpreter or in an object Functions.

When defining functions, if you need a type parameter T, use a parameterized class (or an abstract type).

1.2 Higher order FP (easy)

- Use the definition of *foldLeft* from the slides to concatenate a list of words while separating the words with a space. Try to write the function parameter as concisely as possible.

- Write another definition, which is curried with respect to its last argument, ie, its signature is:

  \[
  \text{def foldLeft}[S, T](s: T, xs: List[S])(f: (T, S) \Rightarrow T): T
  \]

  and repeat the previous exercise.

- Finally, try it with the predefined fold-left operator /: of the class List, whose signature is:

  \[
  \text{def} /:[B](z: B) (op: (B, A) \Rightarrow B): B
  \]
2 Playing with the for expressions (among others)

2.1 Idea
Develop a very simple database engine handling tables (check Movies.scala).

2.2 The tables
A table, instance of class Table is defined by:

- Its name (a string).
- A list of attributes (strings).
- A list of rows, where rows are lists of objects (AnyRef).

Implement this class, adding a simple method display printing the name of the table, its attributes and its rows, one row per line.

2.3 The relational operators
Before defining the methods, add a type Row for the rows, in order to simplify the method signatures. Then implement the following methods:

get(attribute: String, row: Row) This method returns the value of the attribute attribute of the row row.

The next methods all return a new table. This new table doesn’t need a a name, which suggest adding a secondary constructor. These methods can be defined with higher-order methods/functions or with for expressions. We suggest the latter.

project(attributes: List[String]): Table returns the projection of this on the attributes attributes (assuming these attributes exist).

select(predicate: Row => Boolean): Table filters the rows according to the predicate predicate.

product(that: Table): Table returns the cartesian product of this and that.

rename(aliases: Map[String, String]): Table returns a new table with the same data but new attribute names, as defined by aliases.

Here are some elements on maps (class Map, similar to HashMap in Java, except that these maps are immutable - there are also mutable maps).

// creation
val scores = Map("Alice" -> 5, "Bob" -> 4)
// access
scores("Alice") // Java: scores.get("Alice")
// syntactic sugar for a pair
"Alice" -> 5
// a standard pattern
val JacksScore = if (scores.contains("Jack")) scores("Jack") else 0
// which can be implemented as follows
val JacksScore = scores.getOrElse("Jack", 0)
// getting all the values

1 The method index0f of class List returns the index of its argument in this.
scores.values
// getting all the keys
scores.keySet

A key in aliases is an attribute name to change and its value the new attribute name.

equijoin(that: Table, attributeThis: String, attributeThat: String): Table returns a join on this and that based on the equality of the attribute values for attributeThis and attributeThat.

2.4 Eliminating for expressions
Try it on rename, project and join.

2.5 Playing with types
Instead of considering that the elements of a table are of type AnyRef, consider that they are of type T.

2.6 Playing with mutability
Extract all the methods of Table[T] in a trait TableT[T] that does not make no hypothesis on the mutability of subclasses (in terms of modifying the rows) and redefine Table[T] as extending this trait.

Define also another class Relation[T] extending TableT[T] but which is created empty (without a single) row. Add a method insert to add rows.