PRG-w3b: 高階関数

脇田建

2019.10.18

小テスト

Scalaの標準ライブラリが提供する便利な機能

型変数 A

- 型変数の導入: def findFirst[A](…) …
  - 新たな型変数 A を導入する。その変数 A の有効範囲は def の定義の範囲。
    - 型変数 A の意味：「ある型があって、その名前をひとまず A としておこう」、(簡単に)「任意の型Aについて」
- 型構成子の型変数への適用: Seq[A]
  - Seq[A]: 任意の型 A に関して、型構成子 Seq を型Aについて特殊化したもの。
  - Seqの場合 A はSeq が表す配列が要素とするデータの型なので、Seq[A] は、A型のデータを要素とする配列の型と読める。

class Seq[T]が提供する多相関数

- 型変数に依存しない関数群
- def isEmpty: Boolean
- def length: Int
- def size: Int

class Seq[T]が提供する多相関数

- 出現する型変数がTだけで、返り値の型が単純なもの
- def indexOf(T): int
- def forall((T) ⇒ Boolean): Boolean　// ∀
- exists((T) ⇒ Boolean): Boolean　// ∃
- def indexOf(T): int　// Seq(1, 2, 3).indexOf(3) => 2
- def count(p: **(T) ⇒ Boolean**): Int　**高階関数の型**

  // [1,…, 99].count(奇数) => 50

  Range(1, 99).toSeq.count((x) => x % 2 == 0) // 実は toSeq は不要だけど

class Seq[T]が提供する多相関数

- 出現する型変数がTだけで、返り値の型がTを含むもの
- def head: T
- def last: T
- def init: Seq[T]　// Seq(V^n, v) => Seq(V^n)
- def tail: Seq[T]　// Seq(v, V^n) => Seq(V^n)
- def take(Int): Seq[T]　// Seq(V^k, v, …) => Seq(V^k)
- drop(Int): Seq[T]　// Seq(V^k, v, …) => Seq(v, …)

class Seq[T]が提供する多相関数

- **型変数Tが引数にも返り値にも出現するもの**

- def filter(**(T) ⇒ Boolean**): Seq[T]

  // Seq(1, 2, 3, 4, 5).filter(奇数) => Seq(1, 3, 5)

  // Seq(1, 2, 3, 4, 5).filter((n) => n%2 == 1)

- def foldLeft[B](B) ((B, T) ⇒ B): B

foldLeft の直感的な理解

- def foldLeft[B](B) ((B, T) => B): B
- 餌の系列.foldLeft(卵)(パクパク) => 鶏

- **B: 鳥を表す型、T: 餌を表す型**

  パクパク: (B, T) => B

  鳥Bに餌Tをやって少し大きな鳥Tに成長させる

trait Set[T] が提供する関数

- isEmpty: Boolean // 返り値が単相
- empty: Set[A] // 返り値が多相
- contains(A): Boolean // 引数が多相
- diff(GenSet[A]): Set[A] // 引数も返り値も多相 (…[A] => …[A])
- union(GenSet[A]): Set[A]
- map[B]((A) ⇒ B): Set[B]　// Seq(1, 2, 3).map((x: Int) => x.toString)
- subsets(): Iterator[Set[A]]　// Set(1, 2, 3).subsets().foreach(println)

Scala API マニュアル

Scala の API マニュアル

- Macユーザへのお薦め: Dash の利用（強力な検索能力）

- そうではない人は、本家のドキュメントをダウンロードして利用。

- http://scala-lang.org/download/all.html を開き、自分が利用している Scala のバージョン(2.12.7)のページを開き、API DocsのZipファイル(scala-docs-2.12.7.zip)をダウンロードしたあとで展開して利用する。

package/class/trait の
名前で検索

青い◯をクリックして、
Companion Objectの説明
と切り替える。

scala.collection

Seq

Companion object Seq

```
trait Seq[+A] extend…nt, A] with SeqOps[A, Seq, Seq[A]] with
IterableFactoryDefaul…
```

Base trait for sequence collections

A the element type of the collection

Source Seq.scala

Linear Supertypes

Known Subclasses

Type Hierarchy

Filter all members

Abstract Value Members

```
abstract def apply(i: Int): A
```
Get the element at the specified index.

```
abstract def iterator: Iterator[A]
```
Iterator can be used only once

```
abstract def length: Int
```
The length (number of elements) of the sequence.

Concrete Value Members

```
final def ++[B >: A](suffix: IterableOnce[B]): Seq[B]
```
Alias for concat

クラスとCompanionオブジェクト（連れ合いのオブジェクト）

- **class** Int vs **object** Int (scala.Int)
- **class** Seq vs **object** Seq (scala.collection.Seq)
- **class** List vs **object** List (scala.collection.immutable.List)
- **trait** Set vs **object** Set (scala.collection.Set)

class Int vs object Int

- Class Int: Int の代数
  - 算術演算子 (+, -, *, /)、比較演算子 (>, <, ==)、ビット毎演算子
  - min, max, signum
- Object Int: Intに関するシステム情報
  - MaxValue, MinValue
  - toString

class Seq vs object Seq

- class Seq: Seq[T] への操作
- object Seq
  - Seq の作成 (empty[T]: Seq[T]
  - Seq のデータ設定 (fill[T](Int)(⇒T): Seq[T])
    - Seq.fill(5)(3), Seq.fill(100)(math.random)

object List

- empty[A]: List[A]
- fill[A](n: Int) (elem: => A): List[A]
- iterate[A](A, int)((A) ⇒ A)
  / / List.iterate(**List.empty[Int]**, 4)((l: List[Int]) => 0::l)
- foldLeft[B](z: B) (op: (B, A) => B): B
- range[T](T, T) / / List.range(0, 20, 3)

高階関数の利用

高階プログラミング

- 関数を値と扱うプログラミング
  - 引数に関数を渡す
  - 関数の計算結果として関数を返す
  - 関数をデータ構造に含めて保存
  - 関数の計算結果に関数を含める
  - データ構造に保存された関数を取り出す

名前のない関数 (lambda; λ)

- def inc(x: Int) = x + 1
  // 標準的な関数宣言のスタイル

- val inc: **Int => Int** = **x => x + 1**
  // 青谷先生に教わったスタイル

  // 括弧の内側がλ式、括弧は読み易さのため

- val inc = (x: Int) => x + 1
  // 簡略スタイル（引数の型だけを宣言している）

高階関数の例

- def f(g: (Int, Int) => Int) = g(4, 5))
- println(f((x, y) => x + y), f((x, y) => x * y))
- Q: 出力は何だろう？

関数を返す関数

- def add(x: Int): **Int => Int** = {
    def add_y(y: Int) = { x + y }
    add_y
  }

関数を返す関数

- def adder(x: Int): **Int => Int** = {
  def add_y(y: Int) = { x + y }
  add
  }

- **add** なんて中途半端な名前を考えるのも煩わしい

考えてみよう1：関数合成

- 実数上の関数 f, g について関数合成

  compose(f, g) = $g \circ f$ を実装したい

- def compose(f: ?, g: ?): ? = { ? }

考えてみよう1：関数合成

- 実数上の関数 f, g について関数合成

  compose(f, g) = $g \circ f$ を実装したい

- def compose(f: Double => Double,
  g: Double => Double):
  Double => Double = {
  x => g(f(x))
  }

考えてみよう2：関数合成

- 一般の関数 f, g について関数合成

  compose(f, g) = $g \circ f$ を実装したい

- def compose[?](f: ? => ?,
  g: ? => ?):
  ? => ? = { x => g(f(x)) }

考えてみよう2：関数合成

- 一般の関数 f, g について関数合成

  compose(f, g) = $g \circ f$ を実装したい

- def compose[S, T, U](f: S => T, g: T => U): S => U = {

    x => g(f(x))

  }

Curry化

- Scalaの関数定義では複数の引数のリストが書ける。

```
def f(a:A, b:B)(c:C, d:D)(e:E): T = {
    … }
```

- このような関数は段階的に呼び出せる

```
val f1 = f(α, β)
val f2 = f1(γ, δ)
val result = f2(ε)
```

- もちろん、一気に呼び出すこともできる

```
val result = f(α, β)(γ, δ)(ε)
```

Haskell Brooks Curry
(1900-1982)
米国の論理学者

カリー化された関数の宣言の例

- def add1(x: Int, y: Int) = x + y　　// add1(5, 7)
- def add2(x: Int)(y: Int) = x + y　　// add2(5)(7)

  val add5 = add2(5)

  add5(7)
- def add3(x: Int): Int => Int = y => x + y
- def add4: Int => Int => Int = x => y => x + y

既存の関数のカリー化

- 例: def add1(x: Int, y: Int) = x + y
- 自前で頑張る方法
  - def add2(x: Int)(y: Int): Int = add1(x, y)

既存の関数の部分的な適用

- 例: def add1(x: Int, y: Int) = x + y

- add1(5, _:Int)　// (5 + ?) を計算する関数

- add1(_: Int, v)　// (? + 5) を計算する関数

ShapeLib.empty の場合

- def coloredBlock(rows: Int, cols: Int, c: Color): Shape = {
  makeList(rows, makeList(cols, c))
  }
  に対して、以下のように定義できたかも

- val empty = genericEmpty(_: Int, _: Int, Transparent)

class List で定義された関数の利用

- def makeList[T](n: Int, a: T): List[T] = **List.fill**(n)(a)

- def empty(rows: Int, cols: Int): Shape = {
  makeList(rows, makeList(cols, Transparent))
  }

- def size(shape: Shape): (Int, Int) = (shape.**length**, shape(0).**length**)

- def blockCount(shape: Shape): Int = {
  shape.**foldLeft**(0)((count, row) =>
  count + row.**filter**(c => c != Transparent).**length**)
  }