計算機科学第一

11 月 10 日

今週の目標

- 多相的な関数を書けるようになる。

今日の課題 (提出締切は今週金曜日 23 時 59 分 59 秒)

1. モノイドに対する多相的な関数を実装する。

今日のワークフロー

1. GitHub 上の `titech-is-cs115/assignment3` にある課題を終わらせて提出する。

1 モノイド

集合 A と演算 $\cdot : A \times A \to A$ が次の 2 つの条件を満たすとき、ペア $(A, \cdot)$ をモノイドという。

- **結合律:** $(a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \forall a, b, c \in A.$
- **単位元:** $\exists e \in A$ such that $e \cdot a = a \cdot e = a \qquad \forall a \in A.$

以下はモノイドの例である。単位元を明示的に示すために、集合、演算、単位元の三つ組で表現することにする。

- $(\mathbb{Z}, +, 0)$
- $(\mathbb{Z}, \times, 1)$
- $(\mathbb{Z} \cup \{\infty\}, \min, \infty)$
- $(\{\text{true}, \text{false}\}, \wedge, \text{true})$
- $(\{\text{true}, \text{false}\}, \vee, \text{false})$
- $(2^S, \cup, \emptyset)$
- $(2^S, \cap, S)$
- $(S \to S, \circ, \text{id}_S)$ ここで $S \to S$ は集合 S から集合 S への写像の集合、$\circ$ は写像の合成、id_S は S 上の恒等写像を指す。
- $(M_{n,n}, \times, I_n)$ ここで $M_{n,n}$ は実数上の $n \times n$ 行列の集合、$\times$ は行列積、I_n は $n \times n$ の単位行列。

2 Scala でのモノイドの扱い方

Scala では trait を使ってモノイドを表現する。

```
trait Monoid[T] {
  def op(x: T, y: T): T
  def id :T
}
```

trait とはオブジェクトの持つインターフェイスを定義するものである。ここで T は型を表している変数である。モノイド $(\mathbb{Z}, +, 0)$ の定義は以下のようになる。

```
val intAddMonoid = new Monoid[Int] {
  def op(x: Int, y: Int): Int = x + y
  def id = 0
}
```

すると次の様に演算や単位元を利用できる。

```
intAddMonoid.op(2,3)
intAddMonoid.id
```

また、リストのモノイド (リストの結合 ++ を演算に持つモノイド) などモノイド自体を多相的にしたい場合は

```
def listMonoid[A] = new Monoid[List[A]] {
  def op(x: List[A], y: List[A]): List[A] = ...
  def id = ...
}
```

と val ではなく def を使う。このモノイドについて多相的な関数を定義することができる。例えばモノイドの要素 a と b について $a \cdot b \cdot a$ を返す関数 aba は

```
def aba[T](a: T, b: T, m: Monoid[T]) = {
  m.op(m.op(a, b), a)
}
```

と書ける。

```
aba(List(1,2), List(3,4), listMonoid[Int])
```

は

```
List(1,2,3,4,1,2)
```

を返し

```
aba(3, 4, intAddMonoid))
```

は 10 を返す。

3 今日の課題

1. 様々なモノイドの定義を Scala で書け。ソースファイルにコメントで書いてあるモノイドについて定義を書けばよい。

2. モノイド $(A, \cdot_A)$ とモノイド $(B, \cdot_B)$ に対して集合 $A \times B$ の上に自然に演算を定義してモノイドとすることができる。これをモノイドの直積と呼ぶ。二つのモノイドを引数にとりそれらのモノイドの直積を返す関数 `productMonoid[A, B](a: Monoid[A], b: Monoid[B]): Monoid[(A,B)]` を実装せよ。

3. モノイドに対する多相的な関数 `concatenate`、`foldMap`、`pow` を実装せよ。ここで `concatenate` はモノイドの要素からなるリストを受けとり、その要素をリストにおける順番に従いモノイド演算で結合したものを返す関数とする (例: $\texttt{List}(a, b, c) \Rightarrow a \cdot b \cdot c$)。`foldMap` はリストとモノイドへの写像を受け取り、リストの各要素に写像を適用して得られたリストに `concatenate` を適用した結果を返す関数である (例: $(\texttt{List}(a, b, c), f) \Rightarrow f(a) \cdot f(b) \cdot f(c)$)。`pow` はモノイドの要素 a と自然数 n を受け取り、$a^n = \underbrace{a \cdot a \cdot \cdots \cdot a}_{n \text{ times}}$ を返す関数である。ただし、`pow` についてはフィボナッチ数を扱った演習の時のように $O(\log n)$ 回のモノイド演算で計算せよ。

ソースファイルにコメントアウトしてある関数等の型を変更しないこと。また、Section 1で紹介されているモノイドやその他のモノイドを実装してもよい。その場合 pull request のコメントでその旨書くこと。

4 余談: モノイドと並列計算

モノイドの結合法則を利用することで効率的な並列計算が実現できる。結合法則より

$$(((x_1 \cdot x_2) \cdot x_3) \cdot x_4) = ((x_1 \cdot x_2) \cdot (x_3 \cdot x_4))$$

が成り立つ。右辺において $(x_1 \cdot x_2)$ の計算と $(x_3 \cdot x_4)$ の計算は依存関係が無いので、並列計算モデルでは同時に計算できる。そうすると右辺は 2 回のモノイド演算の実行時間で評価できる。一方左辺のように計算する場合には逐次に 3 回のモノイド演算をしないといけないので 3 回分のモノイド演算の時間がかかる。