

オブジェクト指向プログラミングで代数的構造をつくる

中井優

2023 年 3 月 12 日

Tsukuba Computer Mathematics Seminar 2023



- ・ 中井優
- ・ Twitter: [@TOnakai_nakai](https://twitter.com/TOnakai_nakai)



なぜか 今年も B3 枠での発表。
昨年は Minecraft で加算器をつくった。

目次

概要

オブジェクト指向プログラミングとは

オブジェクト指向プログラミングで代数的構造をつくる

整数環

有理数体

多項式環

有理関数体

課題

概要

- ・ オブジェクト指向言語である Java を用いて、代数的構造を実装する。
- ・ オブジェクト指向プログラミングの「メソッド」や「継承」という概念を用いることによって、代数的構造を抽象的に実装することができる。
- ・ 代数的構造を抽象的に実装することで、共通する実装を繰り返さなくて良い。
例えば、ユークリッド整域に対してその商体を定義することで、
個別に有理数体や有理関数体を定義せずに、実装することができる。

オブジェクト指向プログラミングとは

オブジェクト指向プログラミングとは [1]

- ・「オブジェクト」という概念を中心に考えるプログラミングの考え方
- ・「オブジェクト」 = 「それは何か」 + 「それに対する操作」
- ・ C++, Python, Java などの言語は
オブジェクト指向プログラミングをサポートしている.

オブジェクト指向プログラミングで代 数的構造をつくる

Algebra.java

```
interface Monoid { //モノイドの定義
    Monoid multiply(Monoid a); //乗法
    boolean isMultiplIdentityty(); //乗法単位元
}
```

- ・ インタフェースとしてモノイドを定義する。
- ・ インタフェースはクラスみたいなもので、どちらもオブジェクトの設計図。
- ・ モノイドの元に「かける」「単位元かどうか判定する」という操作を定義している。
- ・ これらのオブジェクトに紐付いた「操作」のことを Java ではメソッドという。

Algebra.java

```
interface Group extends Monoid { // 群の定義
    Group multiplyInverse(); // 乗法逆元
    default Monoid divide(Group a) { // 除法は乗法の逆演算
        return multiply(a.multiplyInverse());
    }
}
```

- ・モノイドを継承 (extends) して群を定義する.
- ・継承して作ったクラス (インタフェース) は, 継承元のメソッドを受け継ぐ.
- ・継承 (extends) cf. 実装 (implements).

加法群

Algebra.java

```
interface AddGroup { //加法群の定義
    AddGroup add(AddGroup a); //加法
    boolean isAddIdentity(); //加法単位元
    AddGroup addInverse(); //加法逆元
    default AddGroup subtract(AddGroup a) {
        return add(a.addInverse());
        //減法は加法の逆演算
    }
}
```

- ・モノイドや群の演算は乗法で定義したので、加法群は別に定義する.

Algebra.java

```
interface Ring extends Monoid, AddGroup {}
```

- ・モノイドと加法群を継承するだけで、環が定義できる.

ユークリッド整域

Algebra.java

```
interface EuclideanDomain extends Ring { //ユークリッド整域を定義する
    EuclideanDomain divide(EuclideanDomain a); //除法
    EuclideanDomain remainder(EuclideanDomain a); //剰余

    //GCDを求める
    public static EuclideanDomain gcd(EuclideanDomain a, EuclideanDomain b) {
        if (b .isAddIdentity()) {
            return a;
        } else {
            return gcd(b, a.remainder(b));
        }
    }
}
```

- ・ 環に余り付き除算と GCD 計算を定義するとユークリッド整域になる.

- ・ ユークリッド整域を実装して、整数環をクラスとして定義する.
- ・ インタフェースの実装 (implements) は、もとのメソッドを引き継ぎ、中身を具体的に定義する必要がある.
- ・ つまり、整数環はユークリッド整域の演算を引き継ぎ、それらの演算 (add, multiply など) の中身を具体的に定義しなければならない.
- ・ 一方, subtract, gcd はあらためて定義する必要はない.

Integer.java

```
public class Integer implements EuclideanDomain { //整数環の定義
    private int value; //内部にint型の値を持つ

    public Integer(int n) { //Integer(n)とすることで値をセットする
        this.value = n;
    }

    @Override
    public Monoid multiply(Monoid a) { //乗法
        return new Integer(this.value * ((Integer) a).value);
    }

    //(略)
}
```

Calc.java

```
public class Calc {  
    public static void main(String[] args) {  
        Integer n1 = new Integer(7);  
        Integer n2 = new Integer(2);  
  
        System.out.println("n1 = " + n1);  
        System.out.println("n2 = " + n2);  
        System.out.println("n1 + n2 = " + n1.add(n2));  
        System.out.println("n1 - n2 = " + n1.subtract(n2));  
        System.out.println("n1 * n2 = " + n1.multiply(n2));  
        System.out.println("n1 / n2 = " + n1.divide(n2));  
        System.out.println("n1 % n2 = " + n1.remainder(n2));  
    }  
}
```


実行結果

n1 = 7

n2 = 2

n1 + n2 = 9

n1 - n2 = 5

n1 * n2 = 14

n1 / n2 = 3

n1 % n2 = 1

Algebra.java

```
interface Field extends EuclideanDomain { //体
    Field multiplyInverse(); //乗法逆元

    @Override
    default EuclideanDomain divide(EuclideanDomain a) { //除法
        return (EuclideanDomain)multiply(((Field)a).multiplyInverse());
    }
}
```

- ・ インターフェースとして体を定義する.
- ・ ユークリッド整域に乗法逆元の定義を加える.
- ・ ユークリッド整域から除法の定義を書き換える.

- ・ 型変数 R に対して, R の商体を定義する.
- ・ R はユークリッド整域を継承している.
- ・ クラス自体には体の実装されている.
- ・ (略) の部分には体のメソッドの具体的な処理が書かれている.

QuotientField.java

```
public class QuotientField<R extends EuclideanDomain>
    implements Field {

    public R num; // 分子(numerator)
    public R denom; // 分母(denominator)

    //(略)

}
```

- ・ 型変数 R として整数環をとることで、有理数体が定義できる.

Calc.java

```
public class Calc {
    public static void main(String[] args) {
        QuatientField<Integer> r1
            = new QuatientField<>(new Integer(5), new Integer(6));
        QuatientField<Integer> r2
            = new QuatientField<>(new Integer(3), new Integer(10));

        System.out.println("r1 = " + r1);
        System.out.println("r2 = " + r2);
        System.out.println("r1 + r2 = " + r1.add(r2));
        System.out.println("r1 - r2 = " + r1.subtract(r2));
        System.out.println("r1 * r2 = " + r1.multiply(r2));
        System.out.println("r1 / r2 = " + r1.divide(r2));
    }
}
```

QuotientField.java

$$r1 = 5/6$$

$$r2 = 3/10$$

$$r1 + r2 = 17/15$$

$$r1 - r2 = 8/15$$

$$r1 * r2 = 1/4$$

$$r1 / r2 = 25/9$$

- ・ 型変数 K に対して, K 係数の多項式環を定義する.
- ・ K は体を継承している.
- ・ クラス自体には環が実装されている.
- ・ (略) の部分には環のメソッドの具体的な処理が書かれている.

Poly.java

```
import java.util.ArrayList;

public class Poly<K extends Field>
    extends ArrayList<K> implements EuclideanDomain{
    public String var;

    //(略)
}
```


Calc.java

```
public class Calc {  
    public static void main(String[] args) {  
        QuatientField<Integer> r3  
            = new QuatientField<>(new Integer(0), new Integer(1));  
        QuatientField<Integer> r4  
            = new QuatientField<>(new Integer(1), new Integer(-1));  
        QuatientField<Integer> r5  
            = new QuatientField<>(new Integer(1), new Integer(1));  
  
        Poly<QuatientField<Integer>> p1  
            = new Poly<>(new ArrayList<>(Arrays.asList(r5, r3, r5)), "x");  
        Poly<QuatientField<Integer>> p2  
            = new Poly<>(new ArrayList<>(Arrays.asList(r5, r4)), "x");  
        Poly<QuatientField<Integer>> p3  
            = new Poly<>(new ArrayList<>(Arrays.asList(r5, r5)), "x");  
    }  
}
```

//次のスライドへ続く

Calc.java

// 前のスライドの続き

```
System.out.println("p1 = " + p1);
System.out.println("p2 = " + p2);
System.out.println("p1 + p2 = " + p1.add(p2));
System.out.println("p1 - p2 = " + p1.subtract(p2));
System.out.println("p1 * p2 = " + p1.multiply(p2));
System.out.println("p1 / p2 = " + p1.divide(p2));
System.out.println("p1 % p2 = " + p1.remainder(p2));
    }
}
```

出力結果

$$p1 = 1 + x^2$$

$$p2 = 1 + -1 x^1$$

$$p1 + p2 = 2 + -1 x^1 + x^2$$

$$p1 - p2 = x^1 + x^2$$

$$p1 * p2 = 1 + -1 x^1 + x^2 + -1 x^3$$

$$p1 / p2 = -1 + -1 x^1$$

$$p1 \% p2 = 2$$

- ・ 型変数 R に対して, R の商体を係数体を持つ有理関数体を定義する.
- ・ R はユークリッド整域を継承している.
- ・ クラス自体には体が実装されている.
- ・ 具体的なメソッドは商体を継承している.

RatioFunc.java

```
public class RatioFunc<R extends EuclideanDomain>
    extends QuatientField<Poly<QuatientField<R>>>{

    public RatioFunc(Poly<QuatientField<R>> num, Poly<QuatientField<R>> denom) {
        super(num, denom);
    }
}
```

Calc.java

```
public class Calc {
    public static void main(String[] args) {
        Poly<QuatientField<Integer>> p3 = new Poly<>(new ArrayList<>(Arrays.
            asList(r5, r5)), "x");
        Poly<QuatientField<Integer>> p4 = p2.multiply(p3);
        Poly<QuatientField<Integer>> p5 = p1.multiply(p3);

        System.out.println("p3 = " + p3);
        System.out.println("p4 = " + p4);
        System.out.println("p5 = " + p5);

        RatioFunc<Integer> rf0 = new RatioFunc<>(p4, p5);
        RatioFunc<Integer> rf1 = new RatioFunc<>(p2, p5);
        RatioFunc<Integer> rf2 = new RatioFunc<>(p1, p4);

        //次のスライドに続く
    }
}
```

Calc.java

//前のスライドからの続き

```
System.out.println("rf0 = " + rf0);
```

```
System.out.println("rf1 = " + rf1);
```

```
System.out.println("rf2 = " + rf2);
```

```
System.out.println("rf1 + rf2 = " + rf1.add(rf2));
```

```
System.out.println("rf1 - rf2 = " + rf1.subtract(rf2));
```

```
System.out.println("rf1 * rf2 = " + rf1.multiply(rf2));
```

```
System.out.println("rf1 / rf2 = " + rf1.divide(rf2));
```

```
}
```

```
}
```

出力結果

$$\text{rf0} = (1/2 + 1/-2 x^1)/(1/2 + 1/2 x^2)$$

$$\text{rf1} = (1/4 + 1/-4 x^1)/(1/4 + 1/4 x^1 + 1/4 x^2 + 1/4 x^3)$$

$$\text{rf2} = (1/2 + 1/2 x^2)/(1/2 + 1/-2 x^2)$$

$$\text{rf1} + \text{rf2} = (49/108 + -49/108 x^1 + 49/72 x^2 + 49/216 x^4)/(49/216 + 49/(-216) x^4)$$

$$\text{rf1} - \text{rf2} = (1/-2 x^1 + 1/4 x^2 + 1/-4 x^3)/(1/4 + 1/-4 x^1 + 1/4 x^2 + 1/-4 x^3)$$

$$\text{rf1} * \text{rf2} = 1/(1 + 2 x^1 + x^2)$$

$$\text{rf1} / \text{rf2} = (4 + -8 x^1 + 4 x^2)/(4 + 8 x^2 + 4 x^4)$$

課題




今回は「継承」による代数的構造の実装に重点を置いた。そのため


- ・ 一部の演算が代数的構造で閉じていない。
- ・ それに関連して、型安全性の警告が出る。
- ・ 有理関数の約分に使われる GCD がモニックとは限らないので、有理関数の表記が冗長。

などの課題がある。

本日紹介したコードはすべて GitHub にあります.

<https://github.com/TOnakai-nakai/AlgebraJava>

-  結城浩. Java 言語プログラミングレッスン第3版(上)(下). 電子第1版, SB Creative, 2014.
-  Kredel, H. (2008). Evaluation of a Java Computer Algebra System. In: Kapur, D. (eds) Computer Mathematics. ASCM 2007. Lecture Notes in Computer Science(), vol 5081. Springer, Berlin, Heidelberg.
https://doi.org/10.1007/978-3-540-87827-8_10
-  Taketo Sano. Swift で代数学入門 1～7. Qiita.
<https://qiita.com/taketo1024/items/bd356c59dc0559ee9a0b>
(閲覧 2023-02-20).

-  Ishii, H. (2018). A Purely Functional Computer Algebra System Embedded in Haskell. In: Gerdt, V., Koepf, W., Seiler, W., Vorozhtsov, E. (eds) Computer Algebra in Scientific Computing. CASC 2018. Lecture Notes in Computer Science, vol 11077. Springer, Cham.
https://doi.org/10.1007/978-3-319-99639-4_20

目次

概要

オブジェクト指向プログラミングとは

オブジェクト指向プログラミングで代数的構造をつくる

整数環

有理数体

多項式環

有理関数体

課題