Funtorial Generic Class Programming Pattern

Nicolas Calathus

February 7, 2011

Abstract

This is an attempt to find a proper reusable pattern to support Standard ML like functorial programming style in Java using Java's Generics.

Contents

1 Introduction

This is an attempt to find a proper reusable pattern to support Standard ML like functorial programming style in Java using Java's Generics.
The main advantage of functorial approach is the ability to compose a new class from existing classes through generics. This capability has been almost ignored in Java(generally in most programming language in fact). The key obstacle in Java to support this style was its weak generic feature. In particular, the generics type parameters are not scoped with static property of the generics class.

Often in order to create a new class, it becomes crucial to access parameter class's static information.

But in Java, these generic parameters can be used to constraint the instance property only. Also parametrized class itself does not enforce unique parametrized type for all the instances. since each time, we invoke new A<B>(), we may change B to other B1. As a result, B will not enforce class level restriction with this style.

There are similar restrictions for interface, namely interface can be used only for specifying instance level constraints. It cannot specify a class level constraints, such as static methods, neither it can allow to access class property of parametrized class at run-time. These constraints are too weak to develop a framework to build library which allow to have 'recursive' generic class library, in which applied generics classes can be used as the parameter types for the same generics class from which they generated. This kind of situation can be found in abstract algebra in Mathematics quite often.

In this sample code, we use polynomial ring for the principal means to explore this approach since it is not too trivial and yet it is not too complex. This polynomial ring can be defined from another coefficient ring. So the polynomial ring can be defined as a generic class which takes another ring in the generic type parameter, and create polynomial ring which again implements ring 'signature'(or interface). Therefore, it is possible to use this generated polynomial ring class as a type parameter class for the coefficient ring again. In this way, we can create n-variable polynomial ring from (n-1) variable polynomial ring(starting from another coefficient ring).

2 Approach

These Java generics constraints will enforce certain patterns of (generic) class definition. Also in order to allow to extract proper type information, generic interfaces are required.

This pattern does not address data hiding(abstraction) although it will be addressed in future. The goal of this approach is to avoid casting, and utilizing static type checking, and easy sub-classing/implementation for interface.

3 What is required for future Java

Although this approach will enable to develop 'functorical' generic class in systematic way. But also this indicate the weakness of Java as a programming language. In general, if this style were well established as macro like pattern, it would be easy to incorporate in Java compiler directly. For instance it might be interesting to modify openJDK to support this approach.

4 Some comments on Java generics

There are a few oddity we may face when using this approach. Largely it came form how Java inner (generic) class are implemented. Even though these class are inner class, the class itself does not inherit the class identity from the parent class. Therefore even conceptually it should map to different inner class which are associated to different parametrized class(or just another instance of the parent class), there are treated as the same class. These parametrized class dependency can be achieve only through run-time Type information and actually, the difference cannot be detected at compile type. This is a limitation of using Java. Although conceptually dirty, but there are some advantage for this treatment of inner class. since it will reduce number of different class significantly, it would improve run-time performance.

But the main focus here is not to provide satisfactory static type checking, but to enable to utilize existing class through generics mechanism as functor.

5 Goal of design

This code is essentially a collection of sample programs. So there is no common library to support this approach. The point here is to suggest to design pattern to enable to develop functor like generic class library.

6 Sample top level codes

The most elaborated example here is the creation of a two variable polynomial ring.

This example is a bit messy for repeated generic class declaration, but it can be shorten if we define a subclass for the applied generic class.

here is the top level code:

```
final NumericRing<Float>.CLASS FloatNumeral = NumericRing.CLASS(Float.class);
final NumericPolynomialRing<Float>.CLASS NFloatPolynomialRing = NumericPolynomialRi
{
    final NumericPolynomialRing<Float>.INST nr0 = NFloatPolynomialRing.create(Float
    final NumericPolynomialRing<Float>.INST nr1 = NFloatPolynomialRing.create(2.1f,
    final NumericPolynomialRing<Float>.INST nr2 = NFloatPolynomialRing.create(12f,
    final NumericPolynomialRing<Float>.INST nr3 = NFloatPolynomialRing.create(14f,
    final NumericPolynomialRing<Float>.INST nr4 = NFloatPolynomialRing.create(8f, 1
    final NumericPolynomialRing<Float>.INST nr5 = NFloatPolynomialRing.create(14f,
    final NumericPolynomialRing<Float>.INST nr6 = nr1.mult(nr2);
    System.out.println(">> nr1: "+nr1);
    System.out.println(">> nr2: "+nr2);
    System.out.println(">> nr3: "+nr3);
    System.out.println(">> nr4: "+nr4);
    System.out.println(">> nr5: "+nr5);
    System.out.println(">> nr6: "+nr6);

    final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing<F
```

```
        final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing<F
        final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing<F
        final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing<F
        System.out.println(">> nr1a: "+nr1a);
        System.out.println(">> nr2a: "+nr2a);
        System.out.println(">> nr3a: "+nr3a);
    }
```

output of the execution of this code.

```
>> nr1: (2.1)X^5+(3.0)X^4+(4.0)X^3+(6.5)X^2+(58.0)X+(3.0)
>> nr2: (12.0)X^5+(31.9)X^4+(14.0)X^3+(16.0)X^2+(8.0)X+(13.0)
>> nr3: (14.0)X^3+(16.0)X^2+(8.0)X+(13.0)
>> nr4: (8.0)X+(13.0)
>> nr5: (14.0)X^3+(8.0)X+(13.0)
>> nr6: (25.199999)X^10+(102.99)X^9+(173.09999)X^8+(281.2)X^7+(1024.15)X^6+(2092.5)X^5+(108
>> nr1a: (1.0)X^2*Y^2+(1.0)X*Y^2+(1.0)Y^2+(1.0)X^2*Y+(1.0)X*Y+(1.0)Y+(1.0)X^2+(1.0)X+(1.0)
>> nr2a: (8.0)X*Y^2+(13.0)Y^2+(14.0)X^3*Y+(8.0)X*Y+(13.0)Y+(25.199999)X^10+(102.99)X^9+(173
>> nr3a: (1.0)X^4*Y^4+(2.0)X^3*Y^4+(3.0)X^2*Y^4+(2.0)X*Y^4+(1.0)Y^4+(2.0)X^4*Y^3+(4.0)X^3*Y
```

7 Installation

this project can be imported to netbeans. The project file is under ide/netbeans, you can import from this location.

For running sample, you can select NumericPolynomialRing.java and right click to choose 'run file'.

8 Reference

See some Standard ML papers/books. Old papers/books around late 80's.

A API

```
package com.ncalathus.sample.ring.api;

public interface IAbstractRing {

    public interface CLASS<REP extends INST> {

        REP zero();

        REP unit();

        REP neg_unit();
    }

    public interface INST<REP extends INST> {

        CLASS<REP> CLASS();

        REP add(REP elem);
```

```
        REP subst(REP elem);

        REP mult(REP elem);
    }
}

package com.ncalathus.sample.ring.api;

public interface INumericRing {
    interface CLASS<N extends Number, REP extends INST<N, REP>> extends IAbstractRing.CLASS
        REP create(N n);
    }

    interface INST<N extends Number, REP extends INST<N, REP>> extends IAbstractRing.INST<R

        @Override
        CLASS<N, REP> CLASS();
    }
}

package com.ncalathus.sample.ring.api;

import java.util.Map;

public interface IPolynomialRing {
    interface CLASS<
            C_INST extends IAbstractRing.INST<C_INST>, C_CLASS extends IAbstractRing.CLASS<
            REP extends INST<C_INST, C_CLASS, REP>
        > extends IAbstractRing.CLASS<REP> {

        C_CLASS coefficientRing();

        REP variable(int degree);

        REP create(int degree, C_INST c);

        REP create(final Map<Integer, C_INST> terms);

        REP create(final C_INST... coeffs);

    }

    interface INST<
            C_INST extends IAbstractRing.INST<C_INST>, C_CLASS extends IAbstractRing.CLASS<
            REP extends INST<C_INST, C_CLASS, REP>
        > extends IAbstractRing.INST<REP> {

        @Override
        CLASS<C_INST, C_CLASS, REP> CLASS();

        int degree();

        C_INST coefficient(int degree);

        REP r_mult(C_INST coeff);
    }
```

```
}
```

B Representation

```
package com.ncalathus.sample.ring.rep;

import com.ncalathus.sample.ring.api.*;

// this class allows the all elements of NumericRing to have the same class which follow fr
// Meta class must be used to share the same generic type for static and instance classes.
// Also these static/instance has recuresive references, so outer classes must be used.
// (see NumricRing in the INumericRing_Static<N, NumericRing> for NumericRingStatic)
public class NumericRing<N extends Number> implements IAbstractRing {

    protected final Class<N> _nCls;
    protected final CLASS _class;

    protected NumericRing(final Class<N> nCls) {
        this._nCls = nCls;
        this._class = new CLASS();
    }

    // this is an entry point.
    // all instance of NumericRing are cretated from NumericRingStatic
    // this gurantee, those elements has the same element type.
    public CLASS getCLASS() {
        return _class;
    }

    public static <N extends Number> NumericRing<N>.CLASS CLASS(Class<N> cls) {
        return new NumericRing<N>(cls).getCLASS();
    }

    //
    public class CLASS implements INumericRing.CLASS<N, INST> {
        private final INST _zero;
        private final INST _unit;
        private final INST _neg_unit;

        CLASS() {
            this._zero = new INST(create_zero(_nCls));
            this._unit = new INST(create_unit(_nCls));
            this._neg_unit = new INST(create_neg_unit(_nCls));
        }

        public INST zero() { return _zero; }
        public INST unit() { return _unit; }
        public INST neg_unit() { return _neg_unit; }

        public INST create(N n) {
            return new INST(n);
        }

        final N create_zero(Class<N> nCls) {
            if (nCls.equals(Short.class)) {
                return (N)new Short((short)0);
```

```
        } else if (nCls.equals(Integer.class)) {
            return (N)new Integer(0);
        } else if (nCls.equals(Long.class)) {
            return (N)new Long(0);
        } else if (nCls.equals(Float.class)) {
            return (N)new Float(0);
        } else if (nCls.equals(Double.class)) {
            return (N)new Double(0);
        } else {
            throw new RuntimeException("bug");
        }
    }
    private N create_unit(Class<N> nCls) {
        if (nCls.equals(Short.class)) {
            return (N)new Short((short)1);
        } else if (nCls.equals(Integer.class)) {
            return (N)new Integer(1);
        } else if (nCls.equals(Long.class)) {
            return (N)new Long(1);
        } else if (nCls.equals(Float.class)) {
            return (N)new Float(1);
        } else if (nCls.equals(Double.class)) {
            return (N)new Double(1);
        } else {
            throw new RuntimeException("bug");
        }
    }
    private N create_neg_unit(Class<N> nCls) {
        if (nCls.equals(Short.class)) {
            return (N)new Short((short)(-1));
        } else if (nCls.equals(Integer.class)) {
            return (N)new Integer(-1);
        } else if (nCls.equals(Long.class)) {
            return (N)new Long(-1);
        } else if (nCls.equals(Float.class)) {
            return (N)new Float(-1);
        } else if (nCls.equals(Double.class)) {
            return (N)new Double(-1);
        } else {
            throw new RuntimeException("bug");
        }
    }
}

public class INST implements INumericRing.INST<N, INST> {
    // this is to enforce the interface requirement.
    @Override
    public CLASS CLASS() {
        return _class;
    }

    private final N i;

    INST(final N i) {
        this.i = i;
    }
```

```
@Override
public INST add(final INST elem) {
    return new INST(add(i, elem.i));
}
@Override
public INST subst(final INST elem) {
    return new INST(subst(i, elem.i));
}
@Override
public INST mult(final INST elem) {
    return new INST(mult(i, elem.i));
}

@Override
public String toString() {
    return ""+i;
}

@Override
public boolean equals(Object obj) {
    if (obj instanceof NumericRing.INST) {
        INST inst = (INST)obj;
        return i.equals(inst.i);
    } else {
        return false;
    }
}

@Override
public int hashCode() {
    return i.hashCode();
}

//
private N add(final N n1, final N n2) {
    if (n1 == null || n2 == null) {
        throw new RuntimeException("add for null is not supported");
    }
    if (n1 instanceof Short) {
        final short i1 = (Short)n1;
        final short i2 = (Short)n2;
        return (N)new Short((short)(i1+i2));
    } else if (n1 instanceof Integer) {
        final int i1 = (Integer)n1;
        final int i2 = (Integer)n2;
        return (N)new Integer(i1+i2);
    } else if (n1 instanceof Long) {
        final long i1 = (Long)n1;
        final long i2 = (Long)n2;
        return (N)new Long(i1+i2);
    } else if (n1 instanceof Long) {
        final long i1 = (Long)n1;
        final long i2 = (Long)n2;
        return (N)new Long(i1+i2);
    } else if (n1 instanceof Float) {
        final float i1 = (Float)n1;
        final float i2 = (Float)n2;
```

```
            return (N)new Float(i1+i2);
        } else if (n1 instanceof Double) {
            final double i1 = (Double)n1;
            final double i2 = (Double)n2;
            return (N)new Double(i1+i2);
        } else {
            throw new RuntimeException("bug");
        }
    }
    private N subst(final N n1, final N n2) {
        if (n1 == null || n2 == null) {
            throw new RuntimeException("add for null is not supported");
        }
        if (n1 instanceof Short) {
            final short i1 = (Short)n1;
            final short i2 = (Short)n2;
            return (N)new Short((short)(i1-i2));
        } else if (n1 instanceof Integer) {
            final int i1 = (Integer)n1;
            final int i2 = (Integer)n2;
            return (N)new Integer(i1-i2);
        } else if (n1 instanceof Long) {
            final long i1 = (Long)n1;
            final long i2 = (Long)n2;
            return (N)new Long(i1-i2);
        } else if (n1 instanceof Long) {
            final long i1 = (Long)n1;
            final long i2 = (Long)n2;
            return (N)new Long(i1-i2);
        } else if (n1 instanceof Float) {
            final float i1 = (Float)n1;
            final float i2 = (Float)n2;
            return (N)new Float(i1-i2);
        } else if (n1 instanceof Double) {
            final double i1 = (Double)n1;
            final double i2 = (Double)n2;
            return (N)new Double(i1-i2);
        } else {
            throw new RuntimeException("bug");
        }
    }
    private N mult(final N n1, final N n2) {
        if (n1 == null || n2 == null) {
            throw new RuntimeException("add for null is not supported");
        }
        if (n1 instanceof Short) {
            final short i1 = (Short)n1;
            final short i2 = (Short)n2;
            return (N)new Short((short)(i1*i2));
        } else if (n1 instanceof Integer) {
            final int i1 = (Integer)n1;
            final int i2 = (Integer)n2;
            return (N)new Integer(i1*i2);
        } else if (n1 instanceof Long) {
            final long i1 = (Long)n1;
            final long i2 = (Long)n2;
            return (N)new Long(i1*i2);
```

```
            } else if (n1 instanceof Long) {
                final long i1 = (Long)n1;
                final long i2 = (Long)n2;
                return (N)new Long(i1*i2);
            } else if (n1 instanceof Float) {
                final float i1 = (Float)n1;
                final float i2 = (Float)n2;
                return (N)new Float(i1*i2);
            } else if (n1 instanceof Double) {
                final double i1 = (Double)n1;
                final double i2 = (Double)n2;
                return (N)new Double(i1*i2);
            } else {
                throw new RuntimeException("bug");
            }
        }
    }

    public static void main(String[] args) {
        //NumericRing<Integer> integerRing = new NumericRing<Integer>(Integer.class);
        //final NumericRing<Integer>.CLASS IntegerRing = new NumericRing<Integer>(Integer.c
        final NumericRing<Integer>.CLASS IntegerRing = NumericRing.CLASS(Integer.class);
        final NumericRing<Integer>.INST nr1 = IntegerRing.create(7);
        final NumericRing<Integer>.INST nr2 = IntegerRing.create(9);
        final NumericRing<Integer>.INST nr3 = nr1.mult(nr2);
        System.out.println(">> nr3: "+nr3);
    }
}

package com.ncalathus.sample.ring.rep;

import com.ncalathus.sample.ring.api.IAbstractRing;
import com.ncalathus.sample.ring.api.IPolynomialRing;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class PolynomialRing<C_INST extends IAbstractRing.INST<C_INST>, C_CLASS extends IAbs

    protected final C_CLASS _coeffCls;
    protected CLASS _class;
    protected final String variable;

    protected PolynomialRing(final C_CLASS coeffCls, final String variable) {
        this._coeffCls = coeffCls;
        this.variable = variable;
        initCLASS();
    }

    // this should be overl;oaded by subclass if CLASS is redefined thee.
    protected void initCLASS() {
        this._class = new CLASS();
    }

    // this is an entry point.
```

```
// all instance of NumericRing are cretated from NumericRingStatic
// this gurantee, those elements has the same element type.
//
// this should be overl;oaded by subclass if CLASS is redefined thee.
public CLASS getCLASS() {
    return _class;
}

public static <C_INST extends IAbstractRing.INST<C_INST>, C_CLASS extends IAbstractRing
    return new PolynomialRing<C_INST, C_CLASS>(cls, variable).getCLASS();
}

public class Term {
    final int degree;
    final C_INST c;
    Term(final int degree, final C_INST c) {
        this.degree = degree;
        this.c = c;
    }

    Term r_mult(final C_INST c1) {
        final C_INST c2 = c.mult(c1);
        return new Term(degree, c2);
    }
    Term mult(final Term term) {
        return new Term(degree+term.degree, c.mult(term.c));
    }
    @Override
    public String toString() {
        return "("+c+")"+termString();
    }
    public String termString() {
        if (degree == 0) {
            return "";
        } else if (degree == 1) {
            return variable;
        } else {
            return variable+"^"+degree;
        }
    }
    public String toString(final String vars) {
        return "("+c+")"+termString(vars);
    }
    public String termString(final String vars) {
        if (degree == 0) {
            return vars;
        } else {
            return termString()+((vars.isEmpty())?"":("*"+vars));
        }
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof PolynomialRing.Term) {
            Term term = (Term)obj;
            return degree == term.degree && c == term.c;
        } else {
```

```
            return false;
        }
    }
    @Override
    public int hashCode() {
        return degree+137*c.hashCode();
    }
}

//
public class CLASS implements IPolynomialRing.CLASS<C_INST, C_CLASS, INST> {
    private final INST _zero;
    private final INST _unit;
    private final INST _neg_unit;

    CLASS() {
        this._zero = create(0, _coeffCls.zero());
        this._unit = create(0, _coeffCls.unit());
        this._neg_unit = create(0, _coeffCls.neg_unit());
    }

    public INST zero() { return _zero; }
    public INST unit() { return _unit; }
    public INST neg_unit() { return _neg_unit; }

    @Override
    public C_CLASS coefficientRing() {
        return _coeffCls;
    }

    @Override
    public INST variable(int degree) {
        return new INST(degree, _coeffCls.unit());
    }

    @Override
    public final INST create(int degree, C_INST c) {
        return new INST(degree, c);
    }

    @Override
    public final INST create(final Map<Integer, C_INST> terms) {
        return new INST(new ArrayList<Term>() {{
            for (final Map.Entry<Integer, C_INST> term: terms.entrySet()) {
                add(new Term(term.getKey(), term.getValue()));
            }
        }});
    }

    @Override
    public final INST create(final C_INST... coeffs) {
        return new INST(new ArrayList<Term>() {{
            int degree = coeffs.length-1;
            for (int i = degree; i >= 0; i--) {
                final C_INST c = coeffs[i];
                if (!c.equals(_coeffCls.zero())) {
                    add(new Term(degree-i, c));
```

```
                }
            }
        }});
    }
}

public class INST implements IPolynomialRing.INST<C_INST, C_CLASS, INST> {
    // this is to enforce the interface requirement.
    @Override
    public CLASS CLASS() {
        return _class;
    }

    private final List<Term> terms;

    protected INST(final int degree, final C_INST c) {
        this.terms = new ArrayList<Term>();
        terms.add(new Term(degree, c));
    }
    protected INST(final List<Term> terms) {
        this.terms = normalize(terms);
    }
    protected INST() {
        this.terms = new ArrayList<Term>();
    }

    private List<Term> normalize(List<Term> terms) {
        final Set<Integer> degrees = new HashSet<Integer>();
        for (final Term term: terms) {
            degrees.add(term.degree);
        }
        final List<Integer> sorted_degrees = new ArrayList<Integer>(degrees);
        Collections.sort(sorted_degrees, Collections.reverseOrder());
        final List<Term> norm = new ArrayList<Term>();
        for (final Integer degree: sorted_degrees) {
           final C_INST c0 = sumCoeff(degree, terms);
           if (!c0.equals(_coeffCls.zero())) {
               norm.add(new Term(degree, c0));
           }
        }
        return norm;
    }
    private C_INST sumCoeff(int degree, List<Term> terms) {
        C_INST c = _coeffCls.zero();
        for (final Term term: terms) {
            if (term.degree == degree) {
                c = c.add(term.c);
            }
        }
        return c;
    }

    @Override
    public int degree() {
        if (terms.isEmpty()) {
            return 0;
        }
```

```
        return terms.get(0).degree;
    }
    @Override
    public C_INST coefficient(int degree) {
        for (final Term term: terms) {
            if (term.degree == degree) {
                return term.c;
            }
        }
        return _coeffCls.zero();
    }

    @Override
    public INST add(final INST elem) {
        return new INST(new ArrayList<Term>(){{
            addAll(INST.this.terms);
            addAll(elem.terms);
        }});
    }
    @Override
    public INST subst(final INST elem) {
        return add(elem.r_mult(_coeffCls.neg_unit()));
    }
    @Override
    public INST mult(final INST elem) {
        return new INST(new ArrayList<Term>(){{
            for (final Term term1: terms) {
                for (final Term term2: elem.terms) {
                    add(term1.mult(term2));
                }
            }
        }});
    }

    @Override
    public INST r_mult(final C_INST c) {
        return new INST(new ArrayList<Term>(){{
            for (final Term term: terms) {
                add(term.r_mult(c));
            }
        }});
    }

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder();
        return toString(sb, "");
    }

    public String toString(final StringBuilder sb, final String vars) {
        //final StringBuilder sb = new StringBuilder();
        boolean is_first = true;
        for (final Term term: terms) {
            if (is_first) {
                is_first = false;
            } else {
                sb.append("+");
```

```
            }

            if (term.c instanceof PolynomialRing.INST) {
                final INST inst = (PolynomialRing.INST)term.c;
                final String term0 = term.termString();
                final String vars0 = (vars.isEmpty())?term0:(vars+"*"+term0);
                inst.toString(sb, vars0);
            } else {
                sb.append(term.toString(vars));
            }
        }
        return sb.toString();
    }


    @Override
    public boolean equals(Object obj) {
        if (obj instanceof PolynomialRing.INST) {
            INST inst = (INST)obj;
            return equalTerms(inst.terms);
        } else {
            return false;
        }
    }
    @Override
    public int hashCode() {
        int i = 0;
        for (Term term: terms) {
            i += term.hashCode();
        }
        return i;
    }
    private boolean equalTerms(List<Term> terms0) {
        final int size = terms.size();
        if (terms0.size() != size) {
            return false;
        }
        for (int i = 0; i < size; i++) {
            if (!terms.get(i).equals(terms0.get(i))) {
                return false;
            }
        }
        return true;
    }
}

public static void main(String[] args) {
    {
        //final PolynomialRing<NIntegerRing.INST, NIntegerRing.CLASS>.CLASS IntPolynom
        final PolynomialRing<NIntegerRing.INST, NIntegerRing.CLASS>.CLASS IntPolynomial
        final NIntegerRing.CLASS coeffCls = IntPolynomialRing.coefficientRing();
        final PolynomialRing<NIntegerRing.INST, NIntegerRing.CLASS>.INST nr1 = IntPolyn
        final PolynomialRing<NIntegerRing.INST, NIntegerRing.CLASS>.INST nr2 = IntPolyn
        final PolynomialRing<NIntegerRing.INST, NIntegerRing.CLASS>.INST nr3 = nr1.mult
        System.out.println(">> nr3: "+nr3);
    }
}
```

```
}

package com.ncalathus.sample.ring.rep;

import com.ncalathus.sample.ring.api.IPolynomialRing;
import java.util.HashMap;

public class NumericPolynomialRing<N extends Number> extends PolynomialRing<NumericRing<N>.
    protected NumericPolynomialRing(final NumericRing<N>.CLASS coeffCls, final String varia
        super(coeffCls, variable);
    }
    public class CLASS extends PolynomialRing<NumericRing<N>.INST, NumericRing<N>.CLASS>.CL
        public final INST create(final N... coeffs) {
            return create(new HashMap<Integer, NumericRing<N>.INST>() {{
                int degree = coeffs.length-1;
                for (int i = 0; i <= degree; i++) {
                    final NumericRing<N>.INST c = _coeffCls.create(coeffs[i]);
                    if (!c.equals(_coeffCls.zero())) {
                        put(degree-i, c);
                    }
                }
            }});
        }
    }
    @Override
    protected void initCLASS() {
        this._class = new CLASS();
    }
    @Override
    public CLASS getCLASS() {
        return (CLASS)_class;
    }

    public static <N extends Number> NumericPolynomialRing<N>.CLASS CLASS(final NumericRing
        return new NumericPolynomialRing(nCls, variable).getCLASS();
    }

    //
    //
    //
    static void test_int_1() {
        System.out.println("----------_test_int_1_----------");
        final NumericPolynomialRing<Integer>.CLASS NIntPolynomialRing = NumericPolynomialRi
        final NumericRing<Integer>.CLASS coeffCls = NIntPolynomialRing.coefficientRing();
        {
            final NumericPolynomialRing<Integer>.INST nr1 = NIntPolynomialRing.create(7, co
            final NumericPolynomialRing<Integer>.INST nr2 = NIntPolynomialRing.create(5, co
            final NumericPolynomialRing<Integer>.INST nr3 = nr1.mult(nr2);
            System.out.println(">>_nr1:_"+nr1);
            System.out.println(">>_nr2:_"+nr2);
            System.out.println(">>_nr3:_"+nr3);
        }
        // this looks strange, but it is OK, since subclass deosn ot define CLASS/INST, and
same inner class in byte code level.
        {
            final IntPolyRing.INST nr1 = NIntPolynomialRing.create(7, coeffCls.unit());
            final IntPolyRing.INST nr2 = NIntPolynomialRing.create(5, coeffCls.unit());
            final IntPolyRing.INST nr3 = nr1.mult(nr2);
```

```
        System.out.println(">> nr1: "+nr1);
        System.out.println(">> nr2: "+nr2);
        System.out.println(">>> nr3: "+nr3);
    }
}

static void test_int_2() {
    System.out.println("---------- test_int_2 ----------");
    final NumericPolynomialRing<Integer>.CLASS NIntPolynomialRing = NumericPolynomialRi
    final NumericRing<Integer>.CLASS coeffCls = NIntPolynomialRing.coefficientRing();
    {
        /*
        final NumericPolynomialRing<Integer> npr = new NumericPolynomialRing(NumericRin
        final NumericPolynomialRing<Integer>.INST nr1 = npr.new INST() {{
            final Random rnd = new Random(10);
            for (int i = 0; i < 10; i++) {
                //int degree =
                //add();
            }
        }};
        */
        final NumericPolynomialRing<Integer>.INST nr1 = NIntPolynomialRing.create(2, 3,
        final NumericPolynomialRing<Integer>.INST nr2 = NIntPolynomialRing.create(12, 3
        final NumericPolynomialRing<Integer>.INST nr3 = nr1.mult(nr2);
        System.out.println(">> nr1: "+nr1);
        System.out.println(">> nr2: "+nr2);
        System.out.println(">> nr3: "+nr3);
    }
}
// (compiler) error case
/*
static void test_int_1a() {
    final NumericPolynomialRing<Integer>.CLASS NIntPolynomialRing = NumericPolynomialRi
    final NumericRing<Float>.CLASS coeffCls = NIntPolynomialRing.coefficientRing();
    {
        final NumericPolynomialRing<Integer>.INST nr1 = NIntPolynomialRing.create(7, co
        final NumericPolynomialRing<Integer>.INST nr2 = NIntPolynomialRing.create(5, co
        final NumericPolynomialRing<Integer>.INST nr3 = nr1.mult(nr2);
        System.out.println(">> nr3: "+nr3);
    }
    // strange..
    {
        final IntPolyRing.INST nr1 = NIntPolynomialRing.create(7, coeffCls.unit());
        final IntPolyRing.INST nr2 = NIntPolynomialRing.create(5, coeffCls.unit());
        final IntPolyRing.INST nr3 = nr1.mult(nr2);
        System.out.println(">>> nr3: "+nr3);
    }
}
*/

// this looks type checking is not working as expected..
static void test_int_1b() {
    System.out.println("---------- test_int_1b ----------");
    final NumericPolynomialRing<Integer>.CLASS NIntPolynomialRing = NumericPolynomialRi
    final NIntegerRing.CLASS coeffCls = NIntPolynomialRing.coefficientRing(); // strang
    {
        final NumericPolynomialRing<Integer>.INST nr1 = NIntPolynomialRing.create(7, co
```

```
            final NumericPolynomialRing<Integer>.INST nr2 = NIntPolynomialRing.create(5, co
            final NumericPolynomialRing<Integer>.INST nr3 = nr1.mult(nr2);
            System.out.println(">> nr1: "+nr1);
            System.out.println(">> nr2: "+nr2);
            System.out.println(">> nr3: "+nr3);
        }
        // strange..
        {
            final IntPolyRing.INST nr1 = NIntPolynomialRing.create(7, coeffCls.unit());
            final IntPolyRing.INST nr2 = NIntPolynomialRing.create(5, coeffCls.unit());
            final IntPolyRing.INST nr3 = nr1.mult(nr2);
            System.out.println(">> nr1: "+nr1);
            System.out.println(">> nr2: "+nr2);
            System.out.println(">>> nr3: "+nr3);
        }
    }

    static void test_float_1() {
        System.out.println("---------- test_float_1 ----------");
        final NumericPolynomialRing<Float>.CLASS NFloatPolynomialRing = NumericPolynomialRi
        final NumericRing<Float>.CLASS coeffCls = NFloatPolynomialRing.coefficientRing();
        {
            final NumericPolynomialRing<Float>.INST nr1 = NFloatPolynomialRing.create(7, co
            final NumericPolynomialRing<Float>.INST nr2 = NFloatPolynomialRing.create(5, co
            final NumericPolynomialRing<Float>.INST nr3 = nr1.mult(nr2);
            System.out.println(">> nr1: "+nr1);
            System.out.println(">> nr2: "+nr2);
            System.out.println(">> nr3: "+nr3);
        }
    }

    static void test_float_2() {
        System.out.println("---------- test_float_2 ----------");
        final NumericRing<Float>.CLASS FloatNumeral = NumericRing.CLASS(Float.class);
        final NumericPolynomialRing<Float>.CLASS NFloatPolynomialRing = NumericPolynomialRi
        {
            final NumericPolynomialRing<Float>.INST nr0 = NFloatPolynomialRing.create(Float
            final NumericPolynomialRing<Float>.INST nr1 = NFloatPolynomialRing.create(2.1f,
            final NumericPolynomialRing<Float>.INST nr2 = NFloatPolynomialRing.create(12f,
            final NumericPolynomialRing<Float>.INST nr3 = NFloatPolynomialRing.create(14f,
            final NumericPolynomialRing<Float>.INST nr4 = NFloatPolynomialRing.create(8f, 1
            final NumericPolynomialRing<Float>.INST nr5 = NFloatPolynomialRing.create(14f,
            final NumericPolynomialRing<Float>.INST nr6 = nr1.mult(nr2);
            System.out.println(">> nr1: "+nr1);
            System.out.println(">> nr2: "+nr2);
            System.out.println(">> nr3: "+nr3);
            System.out.println(">> nr4: "+nr4);
            System.out.println(">> nr5: "+nr5);
            System.out.println(">> nr6: "+nr6);

            final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing<F

            //final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing
            //final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing
            //final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing

            final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing<F
```

```
            final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing<F
            final PolynomialRing<NumericPolynomialRing<Float>.INST, NumericPolynomialRing<F
            System.out.println(">> nr1a: "+nr1a);
            System.out.println(">> nr2a: "+nr2a);
            System.out.println(">> nr3a: "+nr3a);
        }

    }

    public static void main(String[] args) {
        /*
        test_int_1();
        test_int_1b();
        test_float_1();
        */
        //test_int_2();
        test_float_2();     }
}
```